# Arm®v8-M Architecture Reference Manual


*DDI0553B.f ID14022019*

## Release information

| Date | Version | Changes |
|---|---|---|
| 14/02/2019 | B.f Non-confidential-EAC | • Sixth release of the v8.0-M manual with integrated v8.1-M material |
| 14/12/2018 | A.j Non-confidential-EAC | • Eighth EAC release |
| 14/12/2018 | B.e Confidential-EAC | • Fifth release of the v8.0-M manual with integrated v8.1-M material |
| 29/06/2018 | A.i Non-confidential-EAC | • Seventh EAC release |
| 29/06/2018 | B.d Confidential-Beta | • Fourth release of the v8.0-M manual with integrated v8.1-M material |
| 31/03/2018 | A.h Non-confidential-EAC | • Sixth EAC release |
| 31/03/2018 | B.c Confidential-Beta | • Third release of the v8.0-M manual with integrated v8.1-M material |
| 15/12/2017 | A.g Non-confidential-EAC | • Fifth EAC release |
| 15/12/2017 | B.b Confidential-Beta | • Second release of the v8.0-M manual with integrated v8.1-M material |
| 29/09/2017 | A.f Non-confidential-EAC | • Fourth EAC release |
| 29/09/2017 | B.a Confidential-Beta | • First release of the v8.0-M manual with integrated v8.1-M material |
| 02/06/2017 | A.e Non-confidential-EAC | • Third EAC release |
| 30/11/2016 | A.d Non-confidential-EAC | • Second EAC release |
| 30/09/2016 | A.c Non-confidential-EAC | • EAC release |
| 28/07/2016 | A.b Non-confidential-Beta | • Beta release |
| 29/03/2016 | A.a Confidential-Beta | • Beta release, limited circulation |

## Armv8-M Architecture Reference Manual



## Proprietary Notice



Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

## Confidentiality Status

## Product Status

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

## Web Address

`http://www.arm.com`

## Contents

# Arm®v8-M Architecture Reference Manual

# Part F Debug Packet Protocols

## Chapter F1 ITM and DWT Packet Protocol Specification



## Glossary

# Preface

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

# About this book

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

This manual has the following parts:

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

**Part B** Describes the architectural rules.

**Part C** Describes the T32 instruction set.

**Part D** Describes the registers.

**Part E** Describes the Armv8-M pseudocode.

**Part F** Describes the packet protocols.

# Using this book

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

**Part A, Armv8-M Architecture Introduction and Overview**

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

Chapter A1 *Introduction*

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

**Part B, Armv8-M Architecture Rules**

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

Chapter B1 *Resets*

Read this for a description of the reset rules.

Chapter B2 *Power Management*

Read this for a description of the power management rules.

Chapter B3 *Programmers' Model*

Read this for a description of the programmers model rules.

Chapter B4 *Floating-point Support*

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

Chapter B5 *Vector Extension*

Read this for a description of the Vector Extension support rules.

Chapter B6 *Memory Model*

Read this for a description of the memory model rules.

Chapter B7 *The System Address Map*

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

Chapter B8 *Synchronization and Semaphores*

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

Chapter B9 *The Armv8-M Protected Memory System Architecture*

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

Chapter B10 *The System Timer, SysTick*

Read this for a description of the system timer rules.

Chapter B11 *Nested Vectored Interrupt Controller*

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

Chapter B12 *Debug*

Read this for a description of the debug rules.

Chapter B13 *Debug and Trace Components*

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

Chapter B14 *The Performance Monitoring Unit Extension*

Read this for a description of the Performance Monitors Extension.

Chapter B15 *Reliability, Availability, and Serviceability (RAS) Extension*

Read this for a description of the Reliability, Availability, and Serviceability (RAS) Extension.

**Part C, Armv8-M Instructions**

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

Chapter C1 *Instruction Set Overview*

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

Chapter C2 *Instruction Specification*

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

**Part D, Armv8-M Registers**

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

Chapter D1 *Register Specification*

Read this for a description of the registers.

**Part E, Armv8-M Pseudocode**

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

Chapter E1 *Arm Pseudocode Definition*

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

Chapter E2 *Pseudocode Specification*

Read this for a description of the pseudocode.

**Part F, Packet Protocols**

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

Chapter F1 *ITM and DWT Packet Protocol Specification*

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

# Conventions

The following sections describe conventions that this book can use:

Typographical conventions.

Signals.

Numbers.

Pseudocode descriptions.

Assembler syntax descriptions.

## Typographical conventions

The typographical conventions are:

*italic*

Introduces special terminology, and denotes citations.

**bold**

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

`monospace`

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

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

SMALLCAPS

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

**Colored text** Indicates a link. This can be:

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

## Signals

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

The signal conventions are:

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

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

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

## Numbers

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

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

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

## Pseudocode descriptions

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

## Assembler syntax descriptions

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

# Additional reading

This section lists relevant publications from Arm and third parties.

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

## Arm publications

- *Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2* (ARM IHI 0031).
- *Arm® Debug Interface Architecture Specification ADIv6.0* (ARM IHI 0074).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *Arm® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.4* (ARM IHI 0064).
- *Embedded Trace Macrocell® ETMv1.0 to ETMv3.5 Architecture Specification* (ARM IHI 0014).
- *Arm®v6-M Architecture Reference Manual* (ARM DDI 0419).
- *Arm®v7-M Architecture Reference Manual* (ARM DDI 0403).
- *Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile* (ARM DDI 0487).
- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile* (ARM DDI587).

## Other publications

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

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

  **Note**

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

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

# Feedback

Arm welcomes feedback on its documentation.

## Feedback on this book

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

- The title.
- The number, DDI0553B.f
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

**Note**

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

# Part A
## Armv8-M Architecture Introduction and Overview

# Chapter A1
# **Introduction**

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

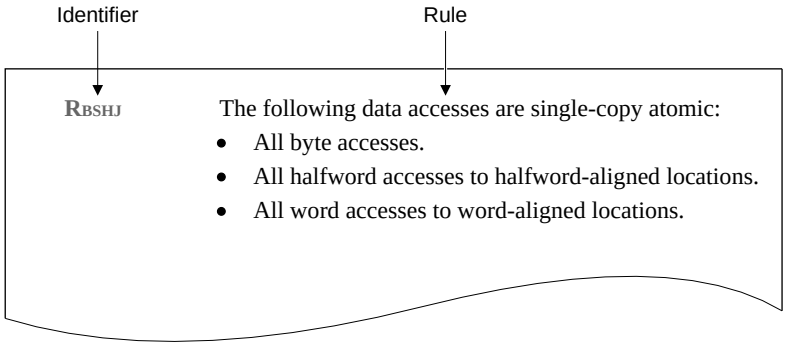
# A1.1 Document layout and terminology

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

## A1.1.1 Structure of the document

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

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




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

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

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

**Note**

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

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

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

- Parts of Chapter B14 *The Performance Monitoring Unit Extension* on page 363.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

- Parts of Part C Armv8-M Instruction Set.
- Part D Armv8-M Register Specification.
- Part E Armv8-M Pseudocode.
- Part F Armv8-M Debug Packet Protocols.

### A1.1.2 Scope of the document

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

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

### A1.1.3 Intended audience

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

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

### A1.1.4 Terminology, phrases

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

**Architecturally visible**

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

**Arm recommends**

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

**Arm strongly recommends**

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

**Finite time**

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

**Permitted**

Allowed behavior.

**Required**

Mandatory behavior.

**Support**

The implementation has implemented a particular feature.

### A1.1.5 Terminology, Armv8-M specific terms

For definitions of Armv8-M specific terms, see the Glossary.

## A1.2 About the Armv8 architecture, and architecture profiles

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

Arm defines three architecture profiles:

**A** Application profile:

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

**R** Real-time profile:

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

**M** Microcontroller profile, described in this manual:

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

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

## A1.3 The Armv8-M architecture profile

The M-profile architecture includes:

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

The simplest Armv8.0-M implementation, without any of the optional extensions, is a Baseline implementation, see A1.4 *Armv8-M variants* on page 51. The Armv8.0-M Baseline offers improvements over previous M-profile architectures in the following areas:

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

### A1.3.1 Security Extension

The Armv8-M architecture introduces a number of new instructions to the M-profile architecture to support asset protection. These instructions are only available to implementations that support the Security Extension, see A1.4 *Armv8-M variants* on page 51.

### A1.3.2 MPU model

The Armv8-M architecture provides a default memory map and permits implementations to include an optional MPU. The optional MPU uses the Protected Memory System Architecture (PMSAv8) and contains improved flexibility in the MPU region definition, see Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

### A1.3.3 Nested Vector Interrupt Controller

The Nested Vector Interrupt Controller (NVIC) is used for integrated interrupt and exception handling and prioritization. Armv8-M increases the number of interrupts that can potentially be supported by the NVIC 480 for external sources, and includes automatic vectoring and priority management, and automatic state preservation. See Chapter B11 *Nested Vectored Interrupt Controller* on page 269.

### A1.3.4 Stack pointers

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

implemented, see B3.8 *Stack pointer* on page 78.

### A1.3.5 The Armv8-M instruction set

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

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

The optional *Floating-point Extension* adds floating-point instructions to the T32 instruction set, see Chapter B4 *Floating-point Support* on page 150.

In an Armv8.1-M implementation a number of non-vector instructions are added to the T32 instruction set, and an implementation might also contain the optional Vector Extensions, see Chapter B5 *Vector Extension* on page 166.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

For more information about the instructions, see Chapter C1 *Instruction Set Overview* on page 420 and Chapter C2, Instruction Specification.

### A1.3.6 Debug

The Armv8-M architecture introduces:

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

In an Armv8.1-M implementation, the optional *Unprivileged Debug Extension* adds support for unprivileged debug.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

For more information about debug, see Chapter B12 *Debug* on page 273 and Chapter B13 *Debug and Trace Components* on page 324.

In an Armv8.1-M implementation, the optional *Performance Monitors Extension* adds support for a Performance Monitor Unit (PMU), see Chapter B14 *The Performance Monitoring Unit Extension* on page 363.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

### A1.3.7 Reliability, Availability, and Serviceability

In an Armv8.1-M implementation, the *Reliability, Availability, and Serviceability (RAS) Extension* adds additional debug support, see Chapter B15 *Reliability, Availability, and Serviceability (RAS) Extension* on page 401. The minimum RAS Extension is mandatory in an Armv8.1-M implementation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

## A1.4 Armv8-M variants

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**DB - The Debug Extension**

> **Note**
>
> For details about the individual features that constitute the Debug Extension, see B12.1 *Debug feature overview* on page 274.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**DIT - Data Independent Timing**

A PE that implements the DIT Extension includes:

- The features that are provided by the Main Extension (M)
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Privileged Execute-Never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**DSP - The Digital Signal Processing Extension.**

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**DSPDE - The DSP Debug Extension**

A PE that implements the DSP Debug Extension includes:

- The features that are provided by the Main Extension (M)
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).
- Data Independent Timing (DIT).
- The Debug Extension (DB).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**FP - The Floating-point Extension**

A PE that implements the Floating-point Extension must implement the Main Extension (M).
The Floating-point Extension supports either single-precision floating-point instructions or both single-precision and double-precision floating-point instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**FPCXT - FPCXT access instructions**

A PE that implements the FPCXT access includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- Low Overhead loops and Branch future (LOB).
- Privileged execute-never (PXN).

- Reliability, Availability, and Serviceability Extension (RAS).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**HP - Half-precision floating-point instructions**

A PE that implements the HP Extension includes:

- The features that are provided by the Main Extension (M).
- Low Overhead loops and Branch future (LOB).
- The Floating-point Extension (FP).
- Reliability, Availability, and Serviceability Extension (RAS).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**LOB - Low Overhead loops and Branch future**

A PE that implements the LOB Extension includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- FPCXT access instructions.
- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**M - The Main Extension**

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

**Note**

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**MPU - The Memory Protection Unit Extension**

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**MVE - M-profile Vector Extension**

**Note**

The Armv8.1-M MVE can also be referred to as Arm® Helium™ technology.

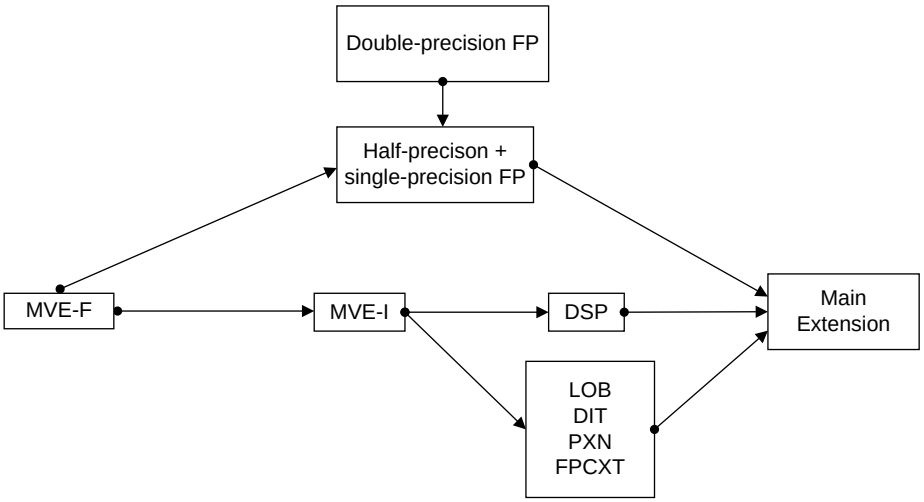
This extension provides operations on various SIMD data types.
It consists of MVE-I (integer) and MVE-F (floating-point).
A PE that implements MVE-F includes:

- Half-precision floating-point instructions (HP).
- The Floating-point Extension (FP).
- MVE-I.

A PE that implements MVE-I includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- (FPCXT) access instructions.
- Low Overhead loops and Branch future (LOB).

- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).
- The DSP Extension (DSP).




*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**PMU - Performance Monitoring Unit**

A PE that implements the PMU Extension includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).

  Some events that are counted by the PMU require additional extensions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**PXN - Privileged eXecute-Never**

A PE that implements the PXN Extension includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Reliability, Availability, and Serviceability Extension (RAS).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**RAS - Reliability, Serviceability, and Availability**

A PE that implements the RAS Extension includes:

- The features that are provided by the Main Extension (M).

- Data Independent Timing (DIT).
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).

  The minimum RAS Extension is mandatory in an Armv8.1-M implementation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

**S - The Security Extension**

**Note**

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**ST - The System Timer Extension**

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**UDE - Unprivileged Debug Extension**

A PE that includes the Unprivileged Debug Extension includes:

- The features that are provided by the Main Extension (M).
- Data Independent Timing (DIT).
- FPCXT access instructions.
- Low Overhead loops and Branch future (LOB).
- Privileged execute-never (PXN).
- Reliability, Availability, and Serviceability Extension (RAS).
- The Debug Extension (DB).
- The Memory Protection Unit (MPU).

The Unprivileged Debug Extension is optional in an Armv8.1-M implementation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

A line below each rule or information statement indicates the architecture version, the extensions that are required for the rule or information statement to apply, and any other notes. Some extensions depend on the implementation of other extensions, for example FP.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

### A1.4.1 Features of Armv8.1-M

The following new features are introduced by Armv8.1-M:

- Registers:
  - DSCEMCR.
  - ERRADDRn.
  - ERRADDR2n.
  - ERRCTRLn.
  - ERRDEVID.

  - ERRFRn.
  - ERRGSRn.
  - ERRIIDR.
  - ERRMISC0n.
  - ERRMISC1n.
  - ERRMISC2n.
  - ERRMISC3n.
  - ERRMISC4n.
  - ERRMISC5n.
  - ERRMISC6n.
  - ERRMISC7n.
  - ERRSTATUSn.
  - FPCXT (payload).
  - LO_BRANCH_INFO (cache).
  - PMU_AUTHSTATUS.
  - PMU_CCFILTR.
  - PMU_CCNTR.
  - PMU_CIDR0.
  - PMU_CIDR1.
  - PMU_CIDR2.
  - PMU_CIDR3.
  - PMU_CNTENCLR.
  - PMU_CNTENSET.
  - PMU_CTRL.
  - PMU_DEVARCH.
  - PMU_DEVTYPE.
  - PMU_EVCNTRn.
  - PMU_EVTYPERn.
  - PMU_INTENCLR.
  - PMU_INTENSET.
  - PMU_OVSCLR.
  - PMU_OVSSET.
  - PMU_PIDR0.
  - PMU_PIDR1.
  - PMU_PIDR2.
  - PMU_PIDR3.
  - PMU_PIDR4.
  - PMU_SWINC.
  - PMU_TYPE.
  - RFSR.
  - VPR.
- MVE instructions:
  - The individual instructions are listed in Chapter C2, Instruction Specification.
- Exception model:
  - New entry to the Stack frame, VPR.
  - Handling of partially executed MVE instructions.

The following Armv8.0-M features are changed by the introduction of the Armv8.1-M architecture:

- The modified registers are:
  - AIRCR.
  - BFSR.
  - CCR.
  - CONTROL.
  - CPACR.

  - CPPWR.
  - DAUTHCTRL.
  - DAUTHSTATUS.
  - DHCSR.
  - DCRSR.
  - DFSR.
  - DWT_CYCCNT.
  - EPSR.
  - FPCAR.
  - FPCCR.
  - FPDSCR.
  - FPSCR.
  - ICSR.
  - ID_DFR0.
  - ID_ISAR0.
  - ID_PFR0.
  - ID_PFR1.
  - MPU_RLAR.
  - MPU_RLAR_An.
  - MVFR0.
  - MVFR1.
  - MVFR2.
  - NSACR.
  - RETPSR (payload).
  - XPSR.

  In addition, the restrictions on access to a number of registers is relaxed to allow a debugger to write to the register when the PE is not in Debug state.

- The Armv8.0-M Floating-point Extension is extended to include half-precision floating-point instructions. These half-precision floating-point instructions are a mandatory part of the Floating-point Extension. These instructions are:

  - VABS.
  - VADD.
  - VCMPE.
  - VCMP.
  - VCVT (between floating-point and fixed-point).
  - VCVT (floating-point to integer).
  - VCVT (integer to floating-point).
  - VCVTA.
  - VCVTM.
  - VCVTN.
  - VCVTP.
  - VCVTR.
  - VDIV.
  - VFMA.
  - VFMS.
  - VFNMA.
  - VFNMS.
  - VLDR.
  - VMAXNM.
  - VMINNM.
  - VMLA.
  - VMLS.
  - VMOV (immediate).

  - VMUL.
  - VNEG.
  - VNMLA.
  - VNMLS.
  - VNMUL.
  - VRINTA.
  - VRINTM.
  - VRINTN.
  - VRINTP.
  - VRINTR.
  - VRINTX.
  - VRINTZ.
  - VSEL.
  - VSQRT.
  - VSTR.
  - VSUB.
- Other modified instructions are:
  - MOV (register).
  - ORR (register).
  - SG.
  - VMOV (half of doubleword register to single general-purpose register) is an alias of VMOV (vector lane to general-purpose register).
  - VMOV (single general-purpose register to half doubleword register) is an alias of VMOV (general-purpose register to vector lane).
  - VMRS.
  - VMSR.

### A1.4.2 Interaction between MVE and the Floating-point Extension in Armv8.1-M

The following architecture features are present in an Armv8.1-M implementation if either or both of MVE and the Floating-point Extension are implemented:

- Registers:
  - S0-S31.
  - CONTROL.{FPCA, SFPA}.
  - FPCCR.
  - FPCAR.
  - FPSCR.
  - MVFR1.
- New and updated instructions:
  - VMOV (register).
  - VINS.
  - VMOVX.
  - VMOV (between general-purpose register and half-precision register).
  - VMOV (between general-purpose registers and single-precision register).
  - VMOV (between two general-purpose register and a doubleword register).
  - VMOV (between two general-purpose registers and two single-precision registers).
  - VMSR, VMRS.
  - VLDM, VSTM, VPUSH, VPOP.
  - VSTR, VLDR.
  - VLSTM, VLLDM.

- Exception model:
  - Lazy and non-lazy stacking of the Floating-point context.
  - Faults that are related to the handling of state in the Floating-point Extension register file, including their corresponding fault status register fields, which are:
    - NOCP UsageFault.
    - MLSPERR MemManage Fault.
    - LSPERR BusFault.
    - LSERR SecureFault, if the Security Extension is implemented.
    - LSPERR Secure Fault, if the Security Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

# Part B
## Armv8-M Architecture Rules

# Chapter B1
# **Resets**

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

B1.1 *Resets, Cold reset, and Warm reset* on page 61.

## B1.1 Resets, Cold reset, and Warm reset

$R_{BDPL}$ There are two resets:

- Cold reset.
- Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CTPC}$ It is not possible to have a Cold reset without also having a Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FNNX}$ On a Cold reset, registers that have a defined reset value contain that value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GTXW}$ On a Warm reset, some debug register control fields that have a defined reset value remain unchanged, but otherwise all registers that have a defined reset value contain that value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{YMHN}$ On a Warm reset, the PE performs the actions that are described by the `TakeReset()` pseudocode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WSZN}$ AIRCR.SYSRESETREQ is used to request a Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HFRS}$ For AIRCR.SYSRESETREQ, the architecture does not guarantee that the reset takes place immediately.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

Chapter B12 *Debug* on page 273.

# Chapter B2
# **Power Management**

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

B2.1 *Power management* on page 63.

# B2.1 Power management

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B2.1.1 The Wait for Event (WFE) instruction

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HDXV}$ When a `WFE` instruction is executed, if the state of the Event register is set, the instruction clears the register and completes immediately.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.13 *Priority model* on page 91.

`WaitForEvent()`.

`SendEvent()`.

## B2.1.2 The Event register

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BPBR}$ The Event register for a PE is set by any of the following:

- Any WFE wakeup event.
- Exception entry.
- Exception return.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{MMZW}$ When the Event register is set, it is an indication that an event has occurred since the register was last cleared, and that the event might require some action by the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CXMT}$ A reset clears the Event register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JFKL}$ The execution of a `WFE` instruction will clear the Event Register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{LNFV}$ Software cannot read, and cannot write to, the Event register directly.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

`SetEventRegister()`

`ClearEventRegister()`

`EventRegistered()`

### B2.1.3 The Wait for Interrupt (WFI) instruction

$R_{HRMJ}$ When a `WFI` instruction is executed, the PE can suspend execution and enter a low-power state. If it does, it remains in that state until it receives a *WFI wakeup event*. When the PE recognizes a `WFI` wakeup event, the `WFI` instruction completes. The following are `WFI` wakeup events:

- A reset.
- Any asynchronous exception at a priority that, ignoring the effect of PRIMASK (so that behavior is as if PRIMASK is 0), would preempt any currently active exceptions.
- An IMPLEMENTATION DEFINED WFI wakeup event.
- If debug is enabled, a debug event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{CGNL}$ Arm recommends that software always uses the `WFI` instruction in a loop.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.13 *Priority model* on page 91.

`WaitForInterrupt()`

## B2.2 Sleep on exit

$R_{JXGW}$ It is IMPLEMENTATION DEFINED whether the `SleepOnExit()` function causes the PE to enter a low-power state during the return from the only active exception and the PE returns to thread mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CMVG}$ The PE enters a low-power state on return from an exception when all the following are true:

- EXC_RETURN.Mode == 1.
- SCR.SLEEPONEXIT== 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WWDW}$ If the sleep-on-exit function is enabled, it is IMPLEMENTATION DEFINED at which point in the exception return process the PE enters a low-power state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LLQF}$ The wakeup events for the sleep-on-exit function are identical to the `WFI` instruction wakeup events.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.13 *Priority model* on page 91.

`SleepOnExit()`

B3.22 *Exception return* on page 115.

# Chapter B3
# **Programmers' Model**

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

# B3.1 PE modes, Thread mode and Handler mode

$R_{CNMS}$ There are two PE modes:

- Thread mode.
- Handler mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CMQP}$ Thread mode is selected on reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.2 *Privileged and unprivileged execution* on page 69.

B3.5.1 *Interrupt Program Status Register (IPSR)* on page 74.

B3.6 *Security states: Secure state, and Non-secure state* on page 76.

## B3.2 Privileged and unprivileged execution

$R_{WVRK}$ **Thread mode**

Execution can be privileged or unprivileged.

**Handler mode**

Execution is always privileged.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{WCFH}$ CONTROL.nPRIV determines whether execution in Thread mode is unprivileged.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SBQF}$ In a PE without the Main Extension, it is IMPLEMENTATION DEFINED whether CONTROL.nPRIV can be set to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JSSW}$ Execution privilege can determine whether a resource is accessible.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{GNSC}$ Privileged execution typically has access to more resources than unprivileged execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.1 *PE modes, Thread mode and Handler mode* on page 68.

## B3.3 Registers

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

**General-purpose registers, all 32-bit**:

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

**Program Counter, 32-bit**:

- R15 is the Program Counter (PC).

**Special-purpose registers**:

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

**Memory-mapped registers**:

All other registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PLRT}$ If the Main Extension is implemented, the LR is set to `0xFFFFFFFF` on Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{QHMH}$ If the Main Extension is not implemented, the LR becomes UNKNOWN on a Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JPCB}$ The PC contains the instruction address of the instruction currently being executed. If an instruction reads the value of the PC, the value returned will increase by 4.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XHHC}$ Except for writes to the CONTROL register, any change to a special-purpose register by a `CPS` or `MSR` instruction is guaranteed:

- Not to affect that `CPS` or `MSR` instruction, or any instruction preceding it in program order.
- To be visible to all instructions that appear in program order after the `CPS` or `MSR`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XPTQ}$ All unallocated or reserved values of fields with allocated values within the memory-mapped registers that are described in this reference manual behave, unless otherwise stated in the register description, in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.
- The encoding causes the field to have no functional effect.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PDJC}$ Reads of registers described as write-only (WO) behave as RES0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

Chapter B7 *The System Address Map* on page 240.

B3.32 *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page 137.

B3.4 *Special-purpose CONTROL register* on page 72.

B3.21 *Stack limit checks* on page 112.

B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

B1.1 *Resets, Cold reset, and Warm reset* on page 61.

Chapter D1 Register Specification.

## B3.4 Special-purpose CONTROL register

$R_{CSPP}$ MRS and MSR instructions can be used to access the CONTROL register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GKVQ}$ Privileged execution can write to the CONTROL register. The PE ignores unprivileged writes to the CONTROL register. All reads of the CONTROL register, regardless of privilege, are allowed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RJMP}$ The architecture requires a Context synchronization event to guarantee visibility of a change to the CONTROL register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HVGB}$ The PE automatically updates CONTROL.SPSEL on exception entry and exception return.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{NMBL}$ CONTROL.SPSEL selects the stack pointer when the PE is in Thread mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

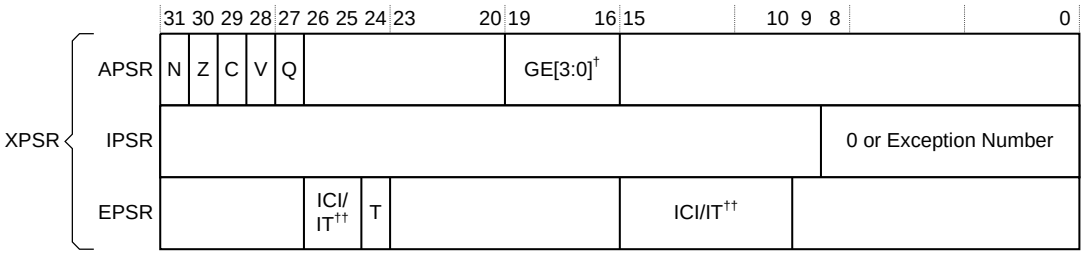
See also:

B3.35 *Context Synchronization Event* on page 148.

*CONTROL, Control Register*.

# B3.5 XPSR, APSR, IPSR, and EPSR

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

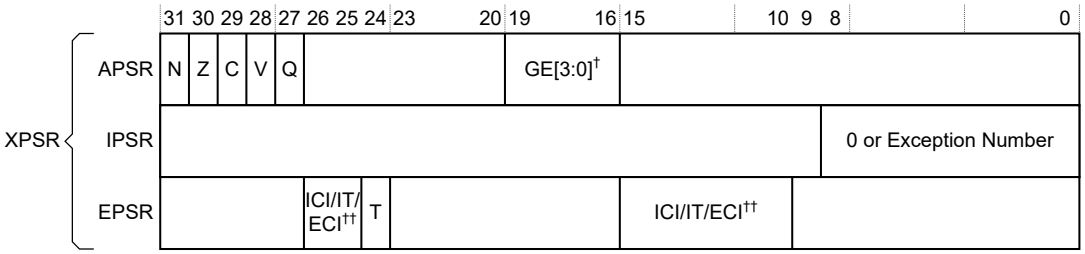



† Reserved if the DSP Extension is not implemented
†† Reserved if the Main Extension is not implemented

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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




† Reserved if the DSP Extension is not implemented
†† Reserved if the Main Extension is not implemented. ECI requires implementing the MVE Extension.

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

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WLFR}$ Arm deprecates using `MSR APSR` without a `_<bits>` qualifier as an alias for `MSR APSR_nzcvq`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.3 *Registers* on page 70.

*APSR, Application Program Status Register*.

B3.5.1 *Interrupt Program Status Register (IPSR)* .

B3.5.2 *Execution Program Status Register (EPSR)* .

### B3.5.1 Interrupt Program Status Register (IPSR)

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

When the PE is in Handler mode:

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

The PE updates the IPSR on exception entry and return.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, Secure state requires S.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CDPK}$ When a CONSTRAINED UNPREDICTABLE instruction is treated as UNDEFINED, an exception is taken. The exception number that is written to the IPSR is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

B3.16 *Function calls from Secure state to Non-secure state* on page 98.

*IPSR, Interrupt Program Status Register*

`BX, BXNS`

### B3.5.2 Execution Program Status Register (EPSR)

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SQLX}$ When EPSR.T is:

**0**: Any attempt to execute any instruction generates:

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, UsageFault requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

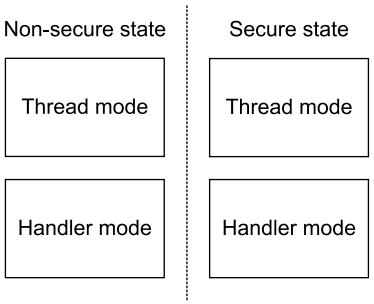
B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

B3.5.2 *Execution Program Status Register (EPSR)* on page 74.

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

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

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




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**.*

See also:

B3.1 *PE modes, Thread mode and Handler mode* on page 68.

B3.2 *Privileged and unprivileged execution* on page 69.

B3.7 *Security states and register banking between Security states* on page 77.

B3.11 *Security states, exception banking* on page 85.

B3.15 *Security state transitions* on page 96.

Chapter B5 *Vector Extension* on page 166.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

# B3.7 Security states and register banking between Security states

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{MKKR}$ This specification uses the following naming convention to identify a Banked register:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

B3.3 *Registers* on page 70.

B3.6 *Security states: Secure state, and Non-secure state* on page 76.

B3.8 *Stack pointer* on page 78.

B7.3 *The System Control Space (SCS)* on page 245.

## B3.8 Stack pointer

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

| Stack | Stack pointer register |
|---|---|
| Main | MPS |
| Process | PSP |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, S is required for MSP_S.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{MQVJ}$ Where an instruction states that the SP is UNPREDICTABLE and SP is used:

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

- If the SP is being written, it is UNKNOWN whether a stack-limit check is applied.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{PWRQ}$ Arm recommends that the Secure stacks be located in Secure memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

B3.6 *Security states: Secure state, and Non-secure state* on page 76.

B3.1 *PE modes, Thread mode and Handler mode* on page 68.

B3.19 *Exception entry, context stacking* on page 103.

B3.30 *Vector tables* on page 134.

B3.3 *Registers* on page 70.

B3.21 *Stack limit checks* on page 112.

## B3.9 Exception numbers and exception priority numbers

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

When AIRCR.BFHFNMINS is 1, faults that target Secure state that are escalated to HardFault are still Secure HardFaults. That is, the value of AIRCR.BFHFNMINS does not affect faults that target Secure state that are escalated to HardFaults. This table row applies to such faults.
If the Security Extension is not implemented exception 7 is reserved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**. Note, S is required for Secure faults.*

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - !**M**. Note, S is required for Secure faults. ST is required for SysTick fault.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

- MemManage Fault.
- BusFault.
- UsageFault.
- SecureFault (if the Security Extension is implemented).
- SVCall.
- DebugMonitor exception.
- PendSV.
- SysTick.
- External Interrupt 0 to *N*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

- SVCall.
- PendSV.
- SysTick.
- External Interrupt 0 to *N*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - !**M**.*

$I_{JQPH}$ All other configurable exceptions can be configured using the NVIC_IPRn.PRI_<n> register fields.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

| Number of priority bits of SHPRIn.PRI_n implemented | Number of configurable Priority numbers | Minimum Priority Number (highest prioirty) | Maximum Priority Number (lowest priority) |
|---|---|---|---|
| 3 | 8 | 0 | 0b11100000 = 224 |
| 4 | 16 | 0 | 0b11110000 = 240 |
| 5 | 32 | 0 | 0b11111000 = 248 |
| 6 | 64 | 0 | 0b11111100 = 252 |
| 7 | 128 | 0 | 0b11111110 = 254 |
| 8 | 256 | 0 | 0b11111111 = 255 |

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - !**M**.*

See also:

B3.11 *Security states, exception banking* on page 85.

B3.12 *Faults* on page 87.

B3.13 *Priority model* on page 91.

SHPR1, SHPR2, SHPR3.

NVIC_IPRn.

`ExecutionPriority()`

# B3.10 Exception enable, pending, and active bits

$I_{QQDG}$ The SHCSR, ICSR, DEMCR, NVIC_IABRn, NVIC_ISPRn contain exception enable, pending, and active fields. STIR can be used to pend exceptions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

- HardFault.
- NMI.
- SVCall.
- PendSV.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**.*

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

- The STIR can pend any Secure or Non-secure interrupt, as follows:

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

- The STIR_NS can pend a Non-secure interrupt, as follows:

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

- The NVIC_ISPRn can pend any Secure or Non-secure interrupt, as follows:

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

- The NVIC_ISPRn_NS can pend a Non-secure interrupt, as follows:

| | Secure state | Non-secure state |
|---|---|---|
| Privileged execution | Can useNVIC_ISPRn_NS to pend a Non-secure interrupt | RES0 |
| Unprivileged execution | BusFault | BusFault |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{TRJJ}$ The following table identifies the fault enable, status and active bits:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B3.11 Security states, exception banking

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, some exceptions require M, S, DebugMonitor exception or ST.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, a UsageFault requires M, Floating-point state requires FP.*

$R_{GVPG}$ If AIRCR.BFHFNMINS == 0, then all Non-secure HardFaults are escalated to Secure HardFaults, and Non-secure pending bits behave as zero for everything except explicit reads and writes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WLGH}$ Where an implementation has two SysTick timers, one in each Security state, each timer targets its owning Security state and not the current Execution state of the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && **ST**.*

$I_{DDKC}$ NMI can be configured to target either Security state, by using AIRCR.BFHFNMINS.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{HGFM}$ BusFault can be configured to target either Security state, by using AIRCR.BFHFNMINS.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MQWN}$ SecureFault always targets Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{WSSL}$ The DebugMonitor exception targets Secure state if the status bit DEMCR.SDME is 1. Otherwise, it targets Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DQLX}$ Each external interrupt, 0-N, targets the Security state that its NVIC_ITNSn.<bit number> dictates.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HXRW}$ When <exception> targets Secure state, the Non-secure view of its priority field, and enabled, pending, and active bits, are RAZ/WI.

<exception> is one of:

- NMI.
- BusFault.
- DebugMonitor.
- External interrupt N.
- In a PE without the Main Extension, and a single instance of the SysTick Timer, SysTick.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, a BusFault exception requires M, a DebugMonitor exception requires DebugMonitor exception.*

$I_{LFHQ}$ Secure software must ensure that when changing the target Security state of an exception, the exception is not pending or active.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.30 *Vector tables* on page 134.

*SHCSR, System Handler Control and State Register*.

## B3.12 Faults

$I_{NHTB}$ There are the following Fault Status Registers:

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

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

There are the following Fault Address Registers:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{LVJW}$ There are the following Fault Status Registers:

- HardFault Status Register HFSR. Present only if the Main Extension is implemented.
- MemManage Fault Status Register MMFSR. Present only if the Main Extension is implemented.
- BusFault Status Register BFSR. Present only if the Main Extension is implemented.
- UsageFault Status Register UFSR. Present only if the Main Extension is implemented.
- SecureFault Status Register SFSR. Present only if the Main Extension is implemented.
- Debug Fault Status Register DFSR. Present only if Halting debug or the Main Extension is implemented.
- Auxiliary Fault Status Register AFSR. The contents of this register are IMPLEMENTATION DEFINED.
- RAS Fault Status Register RFSR. Present only in the Armv8.1-M architecture.

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

There are the following Fault Address Registers:

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

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M**.*

$R_{XMRH}$ Unless otherwise stated, MMFAR is updated only for a MemManage fault on a direct data access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{DDJJ}$ Unless otherwise stated, BFAR is updated only for a BusFault on a data access, a precise fault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BFFR}$ Unless otherwise stated, SFAR is updated only for a SecureFault on a memory access that caused a Security Attribution Unit violation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{FLDT}$ Each fault address register has an associated valid bit. When the PE updates the fault address register, the PE sets the valid bit to 1.

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && !**S**.*

$R_{QPJS}$ If the Security Extension is implemented, it is IMPLEMENTATION DEFINED whether separate BFAR and MMFAR_NS are implemented. If one shared fault address register is implemented, then on a fault that would otherwise update the shared fault address register, if one of the other valid bits is set to one, it is IMPLEMENTATION DEFINED whether:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{GBJF}$ It is IMPLEMENTATION DEFINED whether a separate SFAR and MMFAR_S are implemented. If one secure shared fault address register is implemented, then on a fault that would otherwise update the secure shared fault address register, if the other valid bit for the secure shared fault address register is set to 1, it is IMPLEMENTATION DEFINED whether:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$I_{SCMW}$ Arm strongly recommends that either BFAR is banked between Security states, or, if a single register is implemented, BFAR and the associated FARVALID bits are cleared when changing AIRCR.BFHFNMINS so as not to expose the last accessed address to the other Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

| Exception Number | Exception | | Fault Status Bit |
|---|---|---|---|
| 3 | HardFault | HardFault on Vector table entry read error | HFSR.VECTTBL |
| | | HardFault on fault escalation | HFSR.FORCED |
| | | HardFault on `BKPT` escalation | HFSR.DEBUGEVT |
| 4 | MemManage Fault | MemManage fault on an instruction fetch | MMFSR.IACCVIOL |
| | | MemManage Fault on direct data access | MMFSR.DACCVIOL |
| | | MemManage Fault on context unstacking by hardware. | MMFSR.MUNSTKERR |
| | | MemManage Fault on context stacking by hardware, becauase of a | MMFSR.MSTKERR |

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**. Note, Secure Faults require S.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{XJQC}$ RAS faults can generate BusFaults, and these are recorded in RFSR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

| Exception number | Exception |
|---|---|
| 3 | HardFault |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.31 *Hardware-controlled priority escalation to HardFault* on page 136.

Chapter B12 *Debug* on page 273.

Chapter D1 Register Specification.

# B3.13 Priority model

$I_{CTFJ}$ An exception, other than reset, has the following possible states:

**Active**:
An exception that either:

- Is being handled.
- Was being handled. The handler was preempted by a handler for a higher priority exception.

**Pending**:
An exception that has been generated, but that is not active.

**Inactive**:
The exception has not been generated.

**Active and pending**:
One instance of the exception is active, and a second instance of the exception is pending. Only asynchronous exceptions can be active and pending. Synchronous exceptions are either inactive, pending, or active.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CJDM}$ Lower priority numbers take precedence over higher priority numbers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HLJC}$ When no exception is active and no priority boosting is active, the instruction stream that is executing has a priority number of (maximum supported priority number+1). The instruction stream that is executing can be interrupted by an exception with sufficient priority.

If any exceptions are active the current execution priority is determined by:

1. In a PE with the Main Extension, the calculation of the effect of AIRCR.PRIGROUP on the comparison of BASEPRI to the SHPRn.PRI and NVIC_IPRn values.
2. In a PE with or without the Main Extension applying the effects of PRIMASK.PM and AIRCR.PRIS.
3. In a PE with the Main Extension applying the effects of FAULTMASK.FM.
4. The execution priority is the either:
   - The exception with the lowest priority number.
   - The exception with the lowest priority group value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RKCQ}$ Execution at a particular priority can only be preempted by an exception with a lower group priority value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DPSP}$ In a PE with the Main Extension, BASEPRI and each SHPRn.PRI_n and NVIC_IPRn.PRI_Nn are 8-bit fields that AIRCR.PRIGROUP splits into two fields, a group priority field and a subpriority field:

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

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, Subpriority requires M.*

$R_{CQRV}$ After applying AIRCR.PRIS:

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a Secure exception requires S.*

$R_{KNHG}$ If there are multiple pending exceptions it is IMPLEMENTATION DEFINED whether the AIRCR.PRIGROUP mask is applied to:

- The active tree only.

- The active tree and the pending tree.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

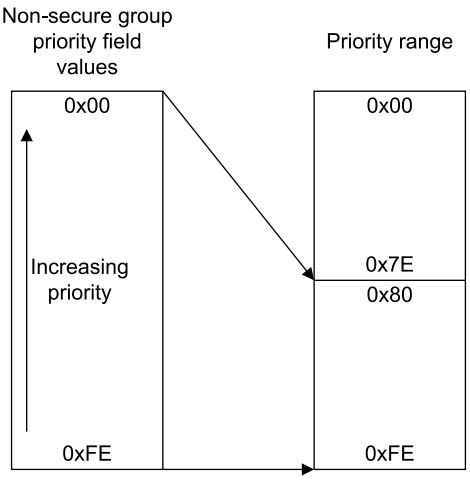
- A has the highest priority.
- B has medium priority.
- C has lowest priority.

Example sequence of events:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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




In this example, the mapping is:

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$I_{WPCP}$ In a PE without the Main Extension but with the Security Extension, when AIRCR.PRIS is set to 1 the Non-secure exception is mapped to the lower half of the priority range, as shown in the table:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && !**M**.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.32 *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page 137.

B3.31 *Hardware-controlled priority escalation to HardFault* on page 136.

`ExceptionPriority()`.

`ExecutionPriority()`.

`ComparePriorities()`.

`RawExecutionPriority()`.

## B3.14 Secure address protection

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{MSNJ}$ *NS-Attr* marks a memory access as Secure or Non-secure.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{VHRL}$ For PE data accesses, NS-Req is equal to the current Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

| NS-Req | Security attribute of the location being accessed | NS-Attr |
|---|---|---|
| Non-secure | X | Non-secure |
| Secure | Non-secure | Non-secure |
| | Secure | Secure |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{NGXH}$ It is not possible to execute Secure code in Non-secure state, or Non-secure code in Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

B3.15 *Security state transitions* on page 96.

B12.3.5 *DAP access permissions* on page 297.

## B3.15 Security state transitions

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{HPPQ}$ The PE will transition from Non-secure to Secure state when all of the following apply:

- The security attribute of the branch target address is Secure and Non-secure callable.
- The branch target address contains an SG which is fetched and executed.
- The whole of the instruction at the branch target address is flagged as Secure and Non-secure callable.
- The execution of the SG instruction does not raise a fault.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

**1**: The target Security state is Secure.

**0**: The target Security state is Non-secure.

Interstating branches are UNDEFINED in Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an INVTRAN SecureFault requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an INVTRAN SecureFault requires M.*

$R_{XNVW}$ If sequential instruction execution crosses from Non-secure memory to Secure memory, then if the Secure memory

entry point contains an SG instruction and the whole of the instruction at the Secure memory entry point is flagged as Secure and Non-secure callable, it is CONSTRAINED UNPREDICTABLE whether:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an INVTRAN SecureFault requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{QVPC}$ When the following conditions are met, the value indicated by the current Secure stack pointer is loaded from memory:

- The SG instruction is executed in Non-secure state.
- Either the SAU or IDAU, or both, indicate that the SG instruction was fetched from Secure memory.
- The PE is executing in Thread mode.

The load of the value indicated by the current Secure stack pointer is performed with the privilege level indicated by CONTROL_S.nPriv and NS-req set to Secure.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **S**.*

$R_{XCXC}$ An INVEP SecureFault is raised if the all of the following are true:

- CCR_S.TRD is set to 1.
- Either, or both, of the following conditions are met:
  - CONTROL_S.SPSEL is 0.
  - The top 31 bits of the value indicated by the current Secure stack pointer loaded from memory matches the top 31 bits of `0xFEFA125A`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **S**.*

See also:

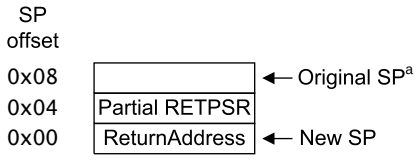
C1.4.7 *Instruction set, interworking and interstating support* on page 439.

Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

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

$R_{GVBB}$ If a BLXNS interstating branch generates a change from Secure state to Non-secure state, then before the Security state change:

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




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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, CONTROL.SFPA requires FP.*

$R_{QVJT}$ Behavior is UNPREDICTABLE when a function call stack frame is not doubleword-aligned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

C1.4.7 *Instruction set, interworking and interstating support* on page 439.

## B3.17 Function returns from Non-secure state

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

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

On detecting a FNC_RETURN value in the PC:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{TFCK}$ If the stack pointer is not 8 byte aligned the behavior is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{DWTF}$ The FNC_RETURN value is:

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

**Bits[31:1]**

This is what identifies the value as an FNC_RETURN value.

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

**0**: Secure state.

**1**: Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && !**M**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, CONTROL.SFPA requires FP.*

$R_{LNFB}$ If the IPSR retrieved from RETPSR is not supported by the PE the value is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

B3.31 *Hardware-controlled priority escalation to HardFault* on page 136.

B3.33 *Lockup* on page 139.

## B3.18 Exception handling

$R_{XGKT}$ In the absence of a specific requirement to take an exception, the architecture requires that pending exceptions are taken within finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KFRF}$ If an exception was pending but is changed to not pending before it is taken, then the architecture permits the exception to be taken but does not require that the exception is taken. If the exception is taken it must be taken before the first Context synchronization event after the exception was changed to not pending.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{YFHR}$ An exception that does not cause lockup sets both:

- The pending bit of its handler, or the pending bit of the HardFault handler, to 1.
- The associated fault status information.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VLDB}$ When a pending exception has a lower group priority value than current execution, including accounting for any priority adjustment by AIRCR.PRIS, the pending exception preempts current execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WBND}$ Preemption of current execution causes the following basic sequence:

1. R0-R3, R12, LR, RETPSR, including CONTROL.SFPA, are stacked.
2. The return address is determined and stacked.
3. Optional stacking of Floating-point context, which might be any one of the following:
   - No stacking or preservation of the Floating-point context.
   - Stacking the basic Floating-point context.
   - Stacking the basic Floating-point context and the additional Floating-point context.
   - Activation of Lazy Floating-point state preservation.
4. LR is set to EXC_RETURN.
5. Optional clearing of Floating-point registers, depending on the Security state transition.
6. The following flags are also cleared:
   - IT State is cleared, if the Main Extension is implemented.
   - CONTROL.FPCA is cleared, if the Floating-point Extension is implemented.
   - CONTROL.SFPA is cleared, if the Floating-point Extension and the Security Extension are implemented.
7. The exception to be taken is chosen, and IPSR Exception is set accordingly. The setting of IPSR Exception to a nonzero value causes the PE to change to Handler mode.
8. CONTROL.SPSEL is set to 0, to indicate the selection of the main stack, dependent on the Security state being targeted.
9. The pending bit of the exception to be taken is set to 0. The active bit of the exception to be taken is set to 1.
10. The Security state is changed to the Security state of the exception that is being activated.
11. The registers are cleared, depending on the transition of the Security state. The registers are divided between the caller and callee registers. If the Security state transition is from Secure to Non-secure state, all the registers are cleared to 0. In all other cases, the caller registers are set to an UNKNOWN value and the callee registers remain unchanged and are not stacked.

12. EPSR.T is set to bit[0] of the exception vector for the exception to be taken.
13. The PC is set to the exception vector for the exception to be taken.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, some steps might require additional extensions.*

$I_{PSGQ}$ The `HandleException()`, `ExceptionEntry()`, `PushStack()`, PushCalleeStack(), `ExceptionTaken()`, and `ActivateException()` pseudocode describes the full exception handling sequence.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BHZN}$ From Armv8.1-M the pseudocode function `HandleExceptionTransitions()` is added.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{NJVF}$ When, during exception entry, the target Security state of an exception differs from the Security state of the memory the exception vector targets:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an INVEP or INVTRAN SecureFault requires M.*

$R_{QLHB}$ The return address is one of the following:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XKDD}$ The least significant bit of the return address from an exception is RES0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.10 *Exception enable, pending, and active bits* on page 83.

B3.13 *Priority model* on page 91.

B3.19 *Exception entry, context stacking* on page 103.

B3.20 *Exception entry, register clearing after context stacking* on page 111.

B3.30 *Vector tables* on page 134.

B3.21 *Stack limit checks* on page 112.

B3.24 *Exceptions during exception entry* on page 120.

Chapter B5 *Vector Extension* on page 166

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

## B3.19 Exception entry, context stacking

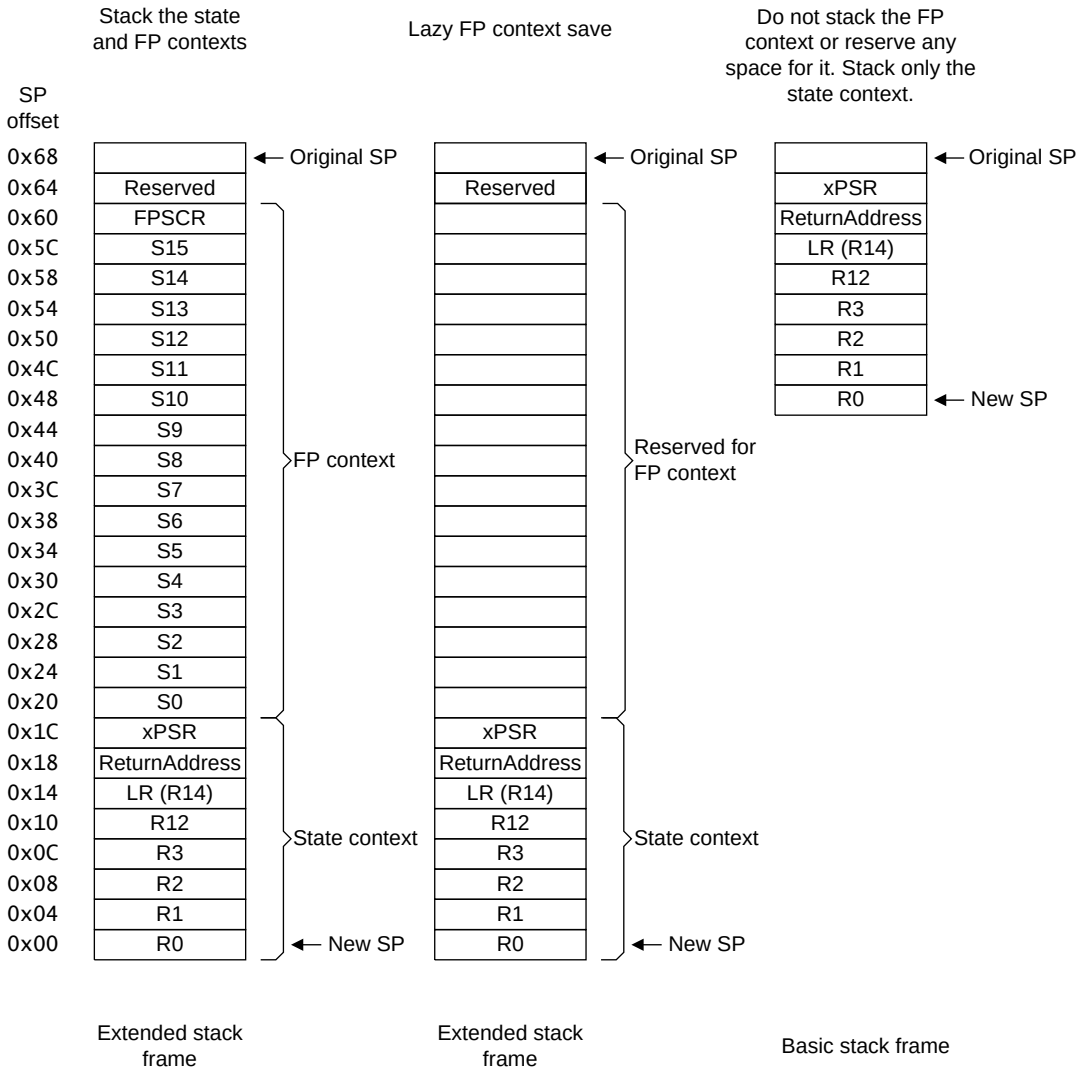
$R_{PWWG}$ On taking an exception, the PE hardware saves *state context* onto the stack that the SP register points to. The state context that is saved is eight 32-bit words:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

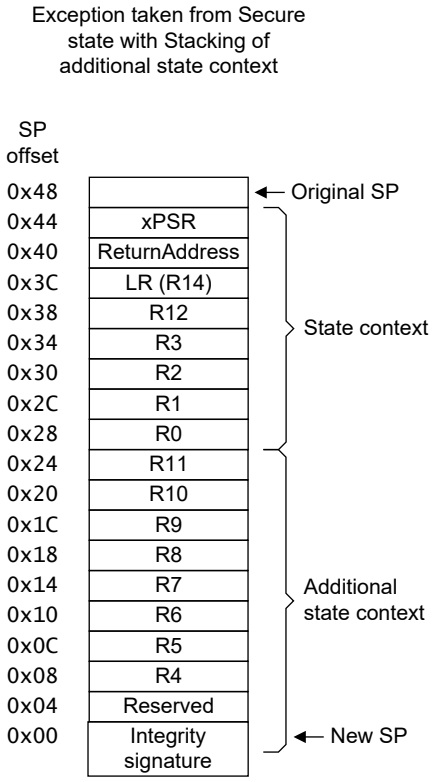
- Stack the Floating-point context.
- Reserve space on the stack for the Floating-point context. This is called *lazy Floating-point context preservation*.




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**. Note, FP is required for the extended stack frame.*

$R_{PLHM}$ In a PE with the Security Extension, on taking an exception, the PE hardware:

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




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{BLQS}$ If a Secure exception is taken from a Secure context of execution, it is IMPLEMENTATION DEFINED whether:

- The additional state context is not stacked, and EXC_RETURN.DCRS is set to 1.
- The additional state context is stacked and EXC_RETURN.DCRS is set to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{KJRL}$ If a higher priority Secure exception occurs during exception entry after the PE has begun stacking the additional state context, but before any exception handler has started execution, in preparation to take a Non-secure exception the PE might:

- Discard the stacking of the additional state context.
- Complete the stacking of the additional state context and the EXC_RETURN.DCRS is set to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

**Non-secure state**

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

**Secure state when CONTROL.FPCA is 0**

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

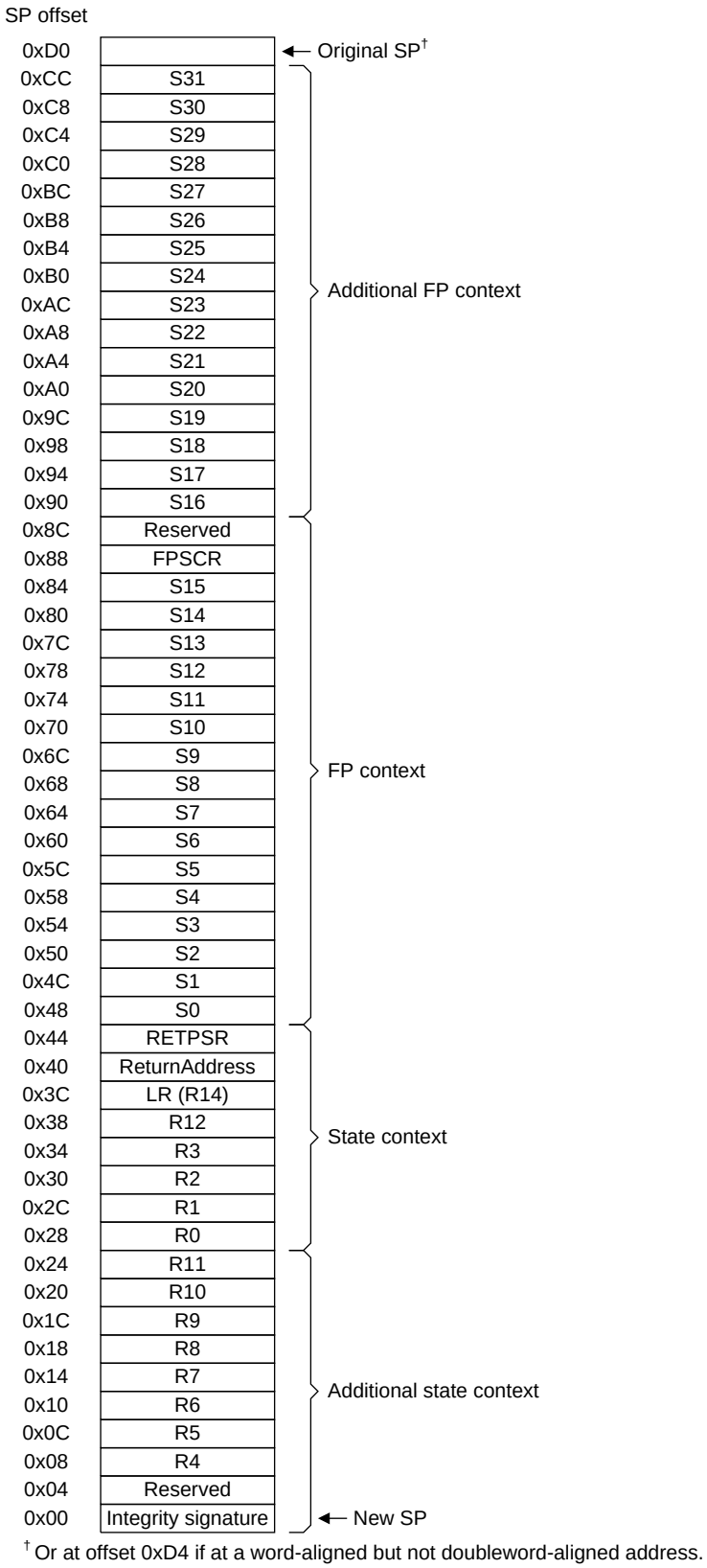
**Secure state when CONTROL.FPCA is 1**

The PE hardware:

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

   If FPCCR_S.TS is 1 and the background state is Secure state when the exception is taken, the PE hardware either stacks both the Floating-point context and additional Floating-point context, or when lazy state preservation is active reserves space on the stack for both the Floating-point context and additional Floating-point context.
3. If exception entry is to Non-secure state, including when a higher priority derived or late-arriving exception targeting Secure state occurs, the PE hardware extends the stack frame, and also saves the additional state context. The PE also performs the exception handling steps common to exception entry.

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




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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && **FP**.*

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

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

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

**0**: The PE did not realign the stack.

**1**: The PE realigned the stack.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, CONTROL.SFPA requires S && FP.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

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

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**. Note, space is reserved for both the Floating-point context and the additional Floating-point context if the Security Extension is implemented.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

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

- If no permission violation is detected, the PE:

1. Saves Floating-point context to the reserved area on the stack, as identified by the FPCAR.
2. Saves the additional Floating-point context if FPCCR.TS and FPCCR.S == 1.
3. Sets FPCCR.LSPACT to 0 to indicate that lazy Floating-point context preservation is no longer active.
4. If the instruction targets Non-secure state the registers are set to an UNKNOWN value. If the instruction targets Secure state the registers are cleared.
5. Processes the floating-point instruction.

- If a permission violation is detected, the PE generates a NOCP UsageFault and does not save Floating-point context to the reserved area on the stack.
- If there is a Security violation or other exception on context stacking the PE will take that exception if the priority of the exception is lower than the execution priority.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

$R_{QLGM}$ A NOCP UsageFault takes precedence over UNDEFINSTR faults for all instructions that fall into the range covered by the `IsCPInstruction()` function.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{HGGX}$ If CP10 is not implemented or disabled, executing an MVE vector instruction generates a NOCP UsageFault.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$R_{NPLD}$ The instruction encoding space `111x 1111 xxxx xxxx xxxx xxxx xxxx xxxx` is part of CP10 and therefore NOCP UsageFaults are prioritized over UNDEFINSTR UsageFaults in the same way as for other co-processor instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{HGRG}$ Some MVE vector instructions exist in the encoding space `111x 1111 xxxx xxxx xxxx xxxx xxxx xxxx`, and these instructions are handled in the same way as the other MVE vector instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

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

- An `SG` instruction fetched from secure memory and executed in Non-secure state clears CONTROL.SFPA to 0.
- A `BXNS` instruction that causes a transition from Secure state to Non-secure state clears CONTROL.SFPA to 0.
- A `BLXNS` instruction that causes a transition from Secure state to Non-secure state preserves the value in CONTROL.SFPA in the FNC_RETURN stack frame and then clears CONTROL.SFPA to 0.
- A valid instruction that loads FNC_RETURN into the PC sets CONTROL.SFPA to the value retrieved from

the FNC_RETURN payload.
- CONTROL.SFPA is saved and restored on exception entry or return in the RETPSR value in the stack frame.
- Exception entry, including tail chaining, clears CONTROL.SFPA to 0.
- If the value of FPCCR.ASPEN is one, then any floating-point instruction (excluding `VLLDM` and `VLSTM`) executed in Secure state sets the value of CONTROL.SFPA to one. If the value of FPCCR.ASPEN is one and the value of CONTROL.SFPA is zero when a floating-point instruction is executed in the Secure state, the FPSCR value is taken from the values set in FPDSCR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

$R_{CGMH}$ In an Armv8.1-M implementation, the value in CONTROL.SFPA is also set automatically by hardware on the following event:

- Saving Secure Floating-point context to a general-purpose register using `VMRS` clears CONTROL.SFPA and VSTR to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** && **S**.*

$I_{GJGL}$ To ensure future compatibility Arm recommends that the value used to seal the top of the stack is `0xFEF5EDA5`. This value has the following properties:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.8 *Stack pointer* on page 78.

B3.20 *Exception entry, register clearing after context stacking* on page 111.

B3.23 *Integrity signature* on page 119.

`PushStack()`.

# B3.20 Exception entry, register clearing after context stacking

$R_{DJRX}$ On exception entry:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**. Note, FP is required for S0-S15 and FPSCR.*

$R_{SNDB}$ On exception entry, including tail-chainging, the PE sets:

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

Otherwise the register values are not changed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{KPZV}$ On exception entry the PE sets R0-R3, R12, APSR, and EPSR to zero regardless of the Security state the exception targets.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **S**.*

$R_{JWBK}$ Register clearing behavior after context stacking is as follows:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

See also:

B3.19 *Exception entry, context stacking* on page 103.

B3.26 *Tail-chaining* on page 123.

# B3.21 Stack limit checks

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M && !S**.*

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

| Security state | Stack | Stack limit registers |
|---|---|---|
| Secure | Main | MSPLIM_S |
| | Process | PSPLIM_S |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S && !M**.*

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

| Stack | Stack limit registers |
|---|---|
| Main | MSPLIM |
| Process | PSPLIM |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M && !S**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M && S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

- Exception entry.
- Tail-chaining from a Secure to a Non-secure exception.
- A function call from Secure code to Non-secure code.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, Secure exceptions and secure code require S.*

$R_{ZLZG}$ On a violation of a stack limit during either exception entry or tail-chaining:

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, A UsageFault requires M.*

$R_{CCSC}$ On a violation of a Secure stack limit during a function call:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, A UsageFault requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

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

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

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

whether stack limit checking is performed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JCCR}$ Updates to the stack pointer by the following instructions are subject to stack limit checking:

- `VLDR (System Register)`.
- `VSTR (System Register)`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$I_{BJHX}$ When an instruction updates the stack pointer, if it results in a violation of the stack limit, it is the modification of the stack pointer that generates the exception, rather than an access that uses the out-of-range stack pointer.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{RRDX}$ CCR.STKOFHFNMIGN controls whether stack limit violations are IGNORED while executing at a requested execution priority that is negative.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XCQL}$ It is UNKNOWN whether a stack limit check is performed on any use of the SP marked as UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QFPF}$ A write to the current stack pointer by an instruction subject to stack limit checking with a value less than the stack limit will generate a STKOF UsageFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DSJN}$ There is no architectural requirement for stack limit checking to be carried out on exception return as the current stack pointer will only increment and will not decrement.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QMRP}$ If an instruction attempts to make an access below the stack limit, it is UNKNOWN whether a store performing a writeback to the current Stack Pointer will generate a STKOF UsageFault where the value written to the current stack pointer is greater than the stack limit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** || **S**.*

$R_{CMBW}$ When a STKOF UsageFault is generated:

- No accesses below the stack limit will be performed.
- It is UNKNOWN whether an access above the stack limit will be performed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.8 *Stack pointer* on page 78.

B3.26 *Tail-chaining* on page 123.

## B3.22 Exception return

$R_{KPSS}$ The PE begins an exception return when both of the following are true:

- The PE is in Handler mode.
- One of the following instructions loads an EXC_RETURN value, `0xFFXXXXXX`, into the PC:
  - A `POP (multiple registers)` or `LDM` that includes loading the PC.
  - An `LDR` with the PC as a destination.
  - A `BX` with any register.
  - A `BXNS` with any register.

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XLCP}$ Behavior is UNPREDICTABLE if any of the following are true and the Security Extension is not implemented:

- EXC_RETURN.S is 1.
- EXC_RETURN.DCRS is 0.
- EXC_RETURN.ES is 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LLBT}$ The following integrity checks on exception return are performed on every exception return:

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

Extension, the INVPC UsageFault targets the Background state.
- In a PE without the Main Extension, a HardFault is generated.

4. If the PE includes the Security Extension, the HardFault targets the Security state that EXC_RETURN.S specifies. If AIRCR.BFHFNMINS is 0 the HardFault targets Secure state, if AIRCR.BFHFNMINS is 1 the exception targets the Security state the exception was returned from.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, some steps require additional extensions, as listed in the rule.*

$R_{HXSR}$ When returning from Non-secure state, EXC_RETURN.ES is treated as zero for all purposes other than raising the *INVER integrity check.*

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{DQLL}$ On returning from Non-secure state, if EXC_RETURN.ES causes an INVER integrity check failure, the subsequent EXC_RETURN.DCRS bit that is presented in the LR on entry to the next exception is permitted to be UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{TLXJ}$ Arm recommends that the subsequent EXC_RETURN.DCRS bit that is presented in the LR on entry to the next exception is not UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JMJC}$ After the EXC_RETURN.ES validation check has been performed on an exception return:

- If EXC_RETURN.ES is 1, EXC_RETURN.SPSEL is written to CONTROL_S.SPSEL.
- If EXC_RETURN.ES is 0, EXC_RETURN.SPSEL is written to CONTROL_NS.SPSEL.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{RPGL}$ On an exception return that successfully returns to the Background state, with no tail-chaining or failed integrity checks, the Security state is set to EXC_RETURN.S.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{CTWL}$ In a PE with the Security Extension, after a successful exception return to the Background state, the PE is in the correct Security state before the next instruction from the background code is executed. This means that in the case where the Background state is Secure state, there is no need for an SG instruction at the exception return address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{RQVB}$ In a PE with the Floating-point Extension register file, on exception entry:

1. EXC_RETURN.FType is saved as the inverse of CONTROL.FPCA.
2. CONTROL.FPCA is then cleared to 0 if it was 1.

On exception return, the inverse of EXC_RETURN.FType is written to CONTROL.FPCA.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{CGML}$ When the following conditions are met on exception return, the PE hardware sets S0-S15 and the FPSCR to 0:

- CONTROL.FPCA is 1.
- FPCCR.CLRONRET is 1.
- If the PE implements the Security Extension FPCCR_S.LSPACT is 0.

If the PE implements the Security Extension and all these fields are 1 on exception return, the PE generates an LSERR SecureFault instead.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**. Note, a SecureFault requires S.*

$R_{ZXTR}$ In an Armv8.1-M implementation, the PE hardware also sets VPR to 0 when:

- CONTROL.FPCA is 1.
- FPCCR.CLRONRET is 1.
- If the PE implements the Security Extension FPCCR_S.LSPACT is 0.

If the PE implements the Security Extension and all these fields are 1 on exception return, the PE generates an LSERR SecureFault instead.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{BVMJ}$ The register clearing described in $R_{CGML}$ only applies if a NOCP UsageFault is not generated due to $R_{FGRC}$.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$R_{XLTP}$ When the following conditions are met on exception return, a NOCP UsageFault is generated:

- CONTROL.FPCA == 1.
- FPCCR.LSPACT == 0.
- FPCCR.CLRONRET == 1.
- Access to CP10 from the Security state of the returning exception, as indicated by EXC_RETURN.ES, is disabled by NSACR, CPACR, OR CPPWR.

The target Security state of the NOCP UsageFault is as follows:

- Secure state, if blocked by NSACR.
- The same Security state as the returning exception as indicated by EXC_RETURN.ES, if blocked by CPACR.
- If the access is blocked by CPPWR, the NOCP Usage fault targets Secure state, if CPPWR.SUS10 ==1. Otherwise, the NOCP UsageFault targets the same Security state as the returning exception as indicated by EXC_RETURN.ES.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**. Note, Secure state requires S.*

$I_{RHNB}$ `IsCPEnabled()` indicates the prioritization if the access is blocked by multiple registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XNNG}$ When the following conditions are met on exception return, the PE generates an LSERR SecureFault:

- EXC_RETURN.FType is 0.
- The stack might contain Secure Floating-point context or Secure lazy floating-point context, that would be unstacked on return. That is, FPCCR_S.LSPACT is 1.
- The return is to Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

$R_{VGGF}$ A check of FPCCR_S.LSPACT, CPACR.CP10, and the relevant fields in NSACR and CPPWR is undertaken prior to unstacking of the floating-point registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{GDVT}$ The floating-point registers are not modified if the checks prior to unstacking fail.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{HNNW}$ If the PE abandons unstacking of the floating-point registers to tail-chain into another exception, then if the Security Extension is implemented, the PE clears to zero any floating-point registers that would have been unstacked.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && **S**.*

$R_{LMNG}$ If the PE abandons unstacking of the floating-point registers to tail-chain into another exception, then if the Security

Extension is not implemented, the floating-point registers that would have been unstacked become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP** && !**S**.*

$R_{HRJH}$ Following completion of the requirements of the EXC_RETURN the PE returns to execution and the following occurs:

- The registers pushed to the stack as part of the exception entry are restored from the stack frame (in accordance with the EXC_RETURN flags).
- APSR, EPSR, and IPSR are restored from RETPSR.
- The PC is set to `ReturnAddress` [31:1]: '0'.
- Bit[0] of the `ReturnAddress` is discarded.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.18 *Exception handling* on page 101.

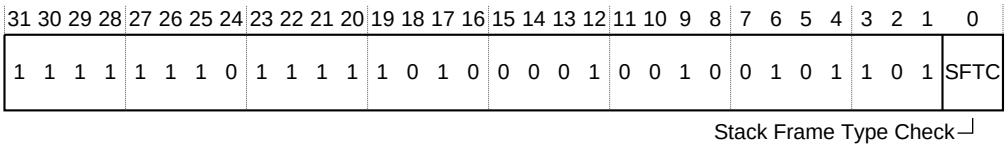
Chapter B5 *Vector Extension* on page 166.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

`ExceptionReturn()`

# B3.23 Integrity signature

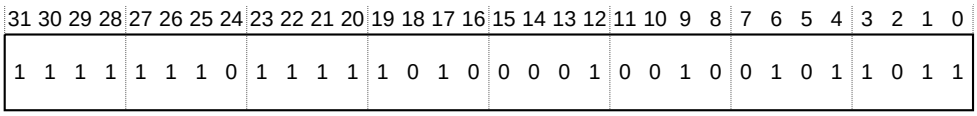
$R_{PHBP}$ In a PE with the Floating-point Extension register file, the integrity signature value is:




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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && **FP**.*

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




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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && !**FP**. Note, a SecureFault requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

B3.19 *Exception entry, context stacking* on page 103.

B3.22 *Exception return* on page 115.

# B3.24 Exceptions during exception entry

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

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NMTT}$ In late-arrival preemption:

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

The pseudocode `DerivedlateArrival()` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

- The stacking of the additional state context is rolled back.
- The stacking of the additional state context is completed and EXC_RETURN is set to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a DebugMonitor Exception requires the DebugMonitor exception.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.13 *Priority model* on page 91.

B3.18 *Exception handling* on page 101.

B3.26 *Tail-chaining* on page 123.

# B3.25 Exceptions during exception return

$I_{KXPV}$ During exception return exceptions can occur, for example asynchronous exceptions, or the exception return might itself cause an exception.

Any exception that occurs during exception return is a *late-arriving exception*.

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TRFM}$ When a late-arriving exception during exception return has a lower priority value than the priority being returned to, the PE takes the late-arriving exception by using tail-chaining.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{MBNG}$ The architecture does not specify the point during exception return at which the PE recognizes the arrival of an asynchronous exception. If a PE recognizes an asynchronous exception after an exception return has completed, there is no opportunity to tail-chain the asynchronous exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MJDN}$ If the priority value of a derived exception during exception return is equal to or higher than the priority being returned to:

- If the derived exception is a DebugMonitor exception, the PE ignores the derived exception.
- Otherwise, the PE escalates the derived exception to HardFault and the escalated exception is tail-chained.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a DebugMonitor Exception requires the DebugMonitor exception.*

$R_{DHFK}$ If the priority value of a derived exception during exception return, after priority escalation if appropriate, is a lower priority value than the execution priority being returned to, the PE uses tail-chaining to take the derived exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.13 *Priority model* on page 91.

B3.22 *Exception return* on page 115.

B3.26 *Tail-chaining* on page 123.

B3.33 *Lockup* on page 139

DebugMonitor exception.

# B3.26 Tail-chaining

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

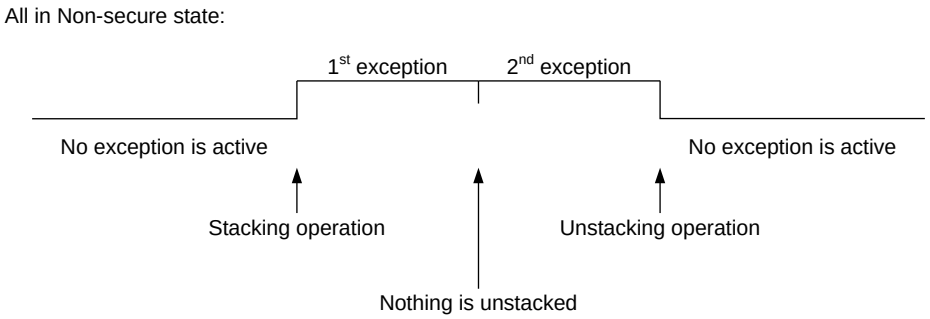
On detecting an EXC_RETURN value in the PC, if there is a pending exception or a derived exception is raised that has a lower prioirty value than the execution priority being returned to, the PE hardware:

1. Does not unstack the stack.
2. Takes the pending exception or derived exception.
   - The PE will tail-chain any pending exception or derived exception on exception return if the pending or derived exception has a lower priority value than the execution priority being returned to.

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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



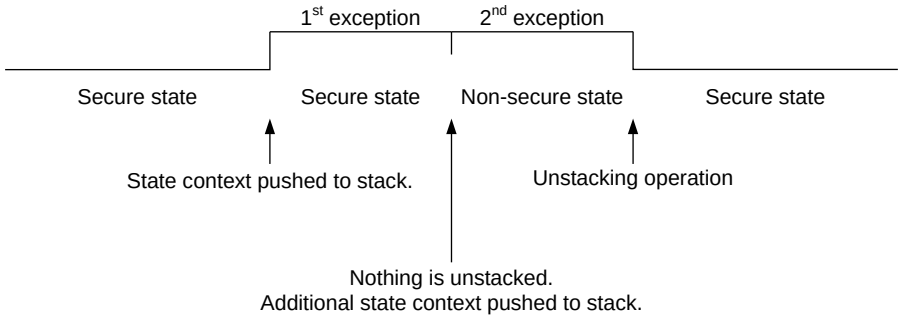

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

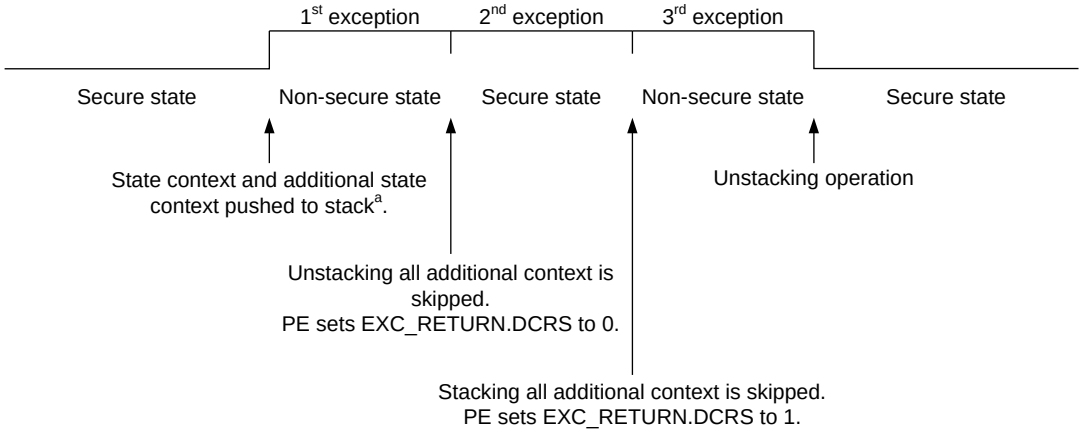
$R_{PXVB}$ When the Background state is Secure state, if tail-chaining causes a change of Security state from Secure to Non-secure, additional context is saved on taking the Non-secure exception if it has not already been saved as indicated by EXC_RETURN.DCRS:

In a PE without the FP Extension:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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




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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.19 *Exception entry, context stacking* on page 103.

B3.25 *Exceptions during exception return* on page 122.

# B3.27 Exceptions, instruction resume, or instruction restart

$R_{PGRC}$ The PE can take an exception during execution of a Load Multiple or Store Multiple instruction, effectively halting the instruction, and resume execution of the instruction after returning from the exception. This is called *instruction resume*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{KRLL}$ The PE can abandon execution of a Load Multiple or Store Multiple instruction to take an exception, and after returning from the exception, restart the Load Multiple or Store Multiple instruction again from the start of the instruction. This is called *instruction restart*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KCMD}$ To support *instruction restart*, singleword load instructions do not update the destination register when the PE takes an exception during execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{NDQT}$ Instructions that the PE can halt to use instruction resume are called *interrupt-continuable instructions*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LGPQ}$ The interrupt-continuable instructions are `LDM`, `LDMDB`, `STM`, `STMDB`, `POP (multiple registers)`, and `Push (multiple registers)`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{RDHK}$ In a PE with the Floating-point Extension, the floating-point interrupt-continuable instructions are `VLDM`, `VLLDM`, `VLSTM`, `VSTM`, `VPOP`, and `VPUSH`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{VFBX}$ Where a fault is taken during the execution of a `VLLDM` instruction the PE abandons the stacking of the Secure floating-point register contents and save the state so that on return from the fault the instruction can be restarted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && **FP**.*

$R_{QWWW}$ It is IMPLEMENTATION DEFINED whether a `VLLDM` and `VLSTM` or instruction aborts or completes when an interrupt occurs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{QVFC}$ When the PE is using instruction resume, EPSR.ICI is set to a non-zero value that is the continuation state of the interrupt-continuable instruction:

- For `LDM`, `LDMDB`, `STM`, `STMDB`, `POP (multiple registers)`, and `PUSH (multiple registers)` instructions, EPSR.ICI contains the number of the first register in the register list that is to be loaded or stored after instruction resume.
- For the floating-point instructions `VLDM`, `VSTM`, `VPOP`, and `VPUSH`, EPSR.ICI contains the number of the lowest numbered doubleword Floating-point Extension register that was not loaded or stored before the PE took the exception.

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, some instructions listed require FP.*

$R_{XFGN}$ Behavior is UNPREDICTABLE if EPSR.ICI contains a valid EPSR.ICI non-zero value and the register number that it contains is either:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{LGJG}$ In addition, for an Armv8.1-M implementation, the PE does not generate an INVSTATE Usage Fault if EPSR.ECI contains a valid nonzero value and the instruction being executed is `CLRM`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{SNRJ}$ In an implementation that includes MVE, the PE does not generate an INVSTATE UsageFault if EPSR.ECI contains a valid ECI value and the instruction is a beatwise MVE instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, Instruction resume requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M**.*

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

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

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

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

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

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

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

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{SGWB}$ When a Load Multiple instruction includes its Base register in its register list, if the PE takes an exception during the instruction:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

# B3.28 Low overhead loops

$R_{FTZN}$ The Armv8.1-M architecture supports low overhead loops using:

- `WLS` - While Loop Start.
- `DLS` - Do Loop Start.
- `LE` - Loop End.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{WGGG}$ An implementation that includes MVE has the following additional instructions that can be used in or in the creation of low overhead loops:

- `WLSTP` - While Loop Start with Tail Predication.
- `DLSTP` - Do Loop Start with Tail Predication.
- `LCTP` - Loop Clear with Tail Predication.
- `LETP` - Loop End with Tail Predication.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **MVE**.*

$R_{FDVJ}$ Instructions within the loop can read and write the loop iteration count, which is in the LR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$I_{GXDN}$ The following is a trivial `memcpy` example which uses the T1 variant of the LE instruction. The T1 variant uses LR:

```
memcpy:
    PUSH    {R0, LR}
    WLS     LR, R2, loopEnd  //R2=size
loopStart:
    LDRSB   R3, [R1], #1    // R1 = srcPtr, R3 = temp reg
    STRB    R3, [R0], #1    // R0 = destPtr
    LE      LR, loopStart

loopEnd:
    POP     {R0, PC}
```

The `WLS` and `LE` instructions cause the loop body to be executed *n* times, where *n* is specified by the value of R2. In this example, the low overhead loop instructions operate as follows:

- If the iteration count that is passed to the `WLS` instruction is nonzero, the loop iteration count is copied to LR. If the iteration count is zero, the `WLS` instruction jumps to the end of the loop.
- If additional iterations of the loop are required when the `LE` instruction is executed (as indicated by the value in LR), the iteration count decrements LR and branches back to the start of the loop. The `LE` instruction also caches the loop branch information in LO_BRANCH_INFO. Subsequent iterations might not be required to re-execute the `LE` instruction.
- If LR indicates that no further iterations are required, the PE branches over the `LE` instruction when execution reaches the last instruction in the body of the loop.

The `LE` T2 variant of the `LE` instruction does not include LR as an argument and can be used where the number of iterations is not known in advance.

The T3 variant of the LE instruction is `LETP`, which is a tail predicated loop.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$I_{CRKM}$ Tail predicated loops can be used if the iteration count is not known in advance. A trivial `memcpy` example of the `LETP` instruction the T3 Variant of the `LE, LETP` instruction is shown here:

```
memcpy:
    PUSH    {R0, LR}
    WLSTP.8     LR, R2, vectLoopEnd  //R2 = element / byte count

vectLoopStart:
    VLDRB.8     Q0, [R1], #16!    // R1 = srcPtr
    VSTRB.8     Q0, [R0], #16!    // R0 = destPtr
    LETP        LR, vectLoopStart

vectLoopEnd:
    POP     {R0, PC}
```

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **MVE**.*

$R_{FZXN}$ The `LE, LETP` instruction caches the loop branch information in LO_BRANCH_INFO.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{HNLF}$ LO_BRANCH_INFO decrements:

- By 1 if FPSCR.LTPSIZE reads as `0b100`.
- By the element width indicated in FPSCR.LTPSIZE, if does not read as `0b100`.

The number of elements is calculated by dividing the vector width, 128, by the element width in FPSCR.LTPSIZE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{VRQV}$ For low overhead loop instructions, LR stores the loop iteration count. For a tail predicated low overhead loop instruction, LR stores the number of vector elements to be processed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **MVE**.*

$R_{ZWFJ}$ The following events update the low overhead loop flags, as indicated by LO_BRANCH_INFO.VALID bit.

| **Event** | LO_BRANCH_INFO |
|---|---|
| Reset | Cleared |
| `LE, LETP` instruction | Conditionally set |
| Execution reaches LO_BRANCH_INFO.END_ADDR | Conditionally cleared |
| `BF, BFX, BFL, BFLX, BFCSEL` instruction | Set |
| Context synchronization event | Cleared |
| `BXNS` or `BLXNS` instruction that cause a Security State transition | Cleared |
| `SG` instuction that causes a transition from Non-secure to Secure state | Cleared |
| Unstacking a FNC_RETURN stack frame | Cleared |
| Any instruction that modifies the PC when LO_BRANCH_INFO.BF is set | Cleared |
| IMPLEMENTATION DEFINED events | Cleared |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**. Note, some rows require S.*

$R_{GHQJ}$ For implementations that include MVE, the architecture permits the architecturally overlapping execution of a vector instruction at the end of the loop with an instruction at the start of the next iteration of the loop, except when:

- The vector instructions at the end of the loop write to LR.
- The instruction at the start of the loop reads or writes to LR.
- Data dependencies between vector instructions are violated.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **MVE**.*

$R_{DMZF}$ When FPSCR.LTPSIZE is not set to `0b100` tail predication is applied according to the value in LR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **MVE**.*

$R_{XZJF}$ When a new floating-point context is created and FPCCR.ASPEN is set to one the PE automatically initializes FPSCR.LTPSIZE to `0b100`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$I_{JLJL}$ When a new floating-point context is created and FPCCR.ASPEN is set to zero it is the responsibility of software to correctly initialize FPSCR.LTPSIZE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{HSNB}$ An INVSTATE UsageFault is raised if an `LE` instruction is executed and FPSCR.LTPSIZE does not read as `0b100`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{RKZN}$ The execution of an implicit or explicit `LE, LETP` instruction is CONSTRAINED UNPREDICTABLE anywhere within an IT block. When the instruction is committed for execution, one of the following occurs:

- An UNDEFINED exception is taken.
- ITSTATE is cleared to 0.
- The instructions are executed as if they had passed the condition code check and the ITSTATE is advanced.
- The instructions execute as NOPs, as if they had failed the condition code check and the ITSTATE is not advanced.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

See also:

B5.6.1 *Loop tail predication* on page 176.

`WLS, DLS, WLSTP, DLSTP`.

`LE, LETP`.

# B3.29 Branch future

$I_{TBQH}$ The Armv8.1-M architecture supports branch future instructions (BF instructions). The BF instruction and its variants are requests to the PE to perform a branch in the future. The variants of the branch future instruction are BF, BFX, BFL, BFLX, and BFCSEL.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$I_{VCGC}$ An example of a BF branch point is as follows:

```
start:
      BFX  branch_point, LR    // Set up BF at branch_point
      ADD r0, r0, r1
      ADD r0, r0, r2
      ADD r0, r0, r3

  branch_point:               // This is the BF branch point
      BX LR                   // Executed if LO_BRANCH_INFO invalid
```

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{SVXL}$ If the last instruction immediately before the BF branch point writes to LR, and a BFL or BFLX set up the BF branch point, then LR is set to the return address, and not to the value that is generated by the instruction at the BF branch point.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{FZVC}$ BF initializes the LO_BRANCH_INFO register to cause a low overhead branch just before execution reaches the specified label, that is the branch point.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{JTGY}$ When BF causes a branch, this branch occurs at the branch point. The instruction after the branch point is not executed if the BF branch point causes a branch.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{JVJF}$ Inserting the BF branch point in the middle of a T32 instruction results in one of the following CONSTRAINED UNPREDICTABLE behaviors:

- It executes as a NOP.
- It raises an UNDEFINED instruction fault.
- It executes normally and the branch that is associated with the BF instruction is taken.
- It executes normally and the branch that is associated with the BF instruction is not taken.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{VGJR}$ If the BF branch point is in an IT block, and it does not immediately precede the last instruction in the IT block, then the following CONSTRAINED UNPREDICTABLE behaviors can result:

- The instruction executes normally and the branch that is associated with the BF instruction is not taken. The BF instruction can be treated as a NOP.
- The instruction before the BF branch point raises an UNDEFINED instruction fault.
- ITSTATE is cleared to 0.
- Taking the branch that is associated with the BF instruction causes ITSTATE to become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{HLVZ}$ If a BF branch point is within an IT block, the branch that was created by the BF instruction is not affected by the

IT condition.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{JFHP}$ When executing in Handler mode, `BF` instructions that attempt to cause a branch to EXC_RETURN behave as NOPs.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{BGSF}$ In an implementation that includes the Security Extension, `BF` instructions that attempt to cause a branch to FNC_RETURN behave as NOPs.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && **S**.*

$R_{MCKJ}$ Taking a branch that is created by the BF instruction clears ITSTATE to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

See also:

Chapter C1 *Instruction Set Overview* on page 420.

B3.28 *Low overhead loops* on page 129.

C1.3.5 *ITSTATE* on page 431.

`BF, BFX, BFL, BFLX, and BFCSEL`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

# B3.30 Vector tables

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

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

If the PE supports configurability of each vector table base:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - !**S**.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{WPRT}$ A vector table contains both:

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

The *exception number* defines the order of entries.

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{PLSB}$ Arm recommends that it is ensured that the vector table and VTOR are aligned so that the entry for the highest taken exception falls within the natural alignment of the table, and at a minimum that the vector table is 128 byte aligned. A PE might impose further restrictions on the VTOR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

- The PE enters Lockup at the priority of the original exception.
- The original exception transitions from the pending to the active state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, SAU and IDAU require M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B9.4 *IMPLEMENTATION DEFINED Attribution Unit (IDAU)* on page 265.

B9.3 *Security attribution unit (SAU)* on page 264.

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.5.2 *Execution Program Status Register (EPSR)* on page 74.

## B3.31 Hardware-controlled priority escalation to HardFault

$R_{GTNS}$ An interrupt is escalated to HardFault in the following way:

- If the priority value of the current execution is greater than the priority value of the fault or interrupt, the fault or interrupt is taken.
- If the priority value of the fault or interrupt is greater than the priority value of the current execution the fault or interrupt is escalated to HardFault.
- If the HardFault cannot be taken the PE enters Lockup.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, DebugMonitor exception requires M.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

DebugMonitor exception.

B3.33 *Lockup* on page 139.

B3.11 *Security states, exception banking* on page 85.

## B3.32 Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting

$I_{BNJG}$ In a PE with the Main Extension, the PRIMASK, FAULTMASK, and BASEPRI registers can be used as follows. A PE without the Main Extension implements PRIMASK, but does not implement FAULTMASK and BASEPRI.

**PRIMASK**

In a PE without the Security Extension:

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

In a PE with the Security Extension:

- Setting PRIMASK_S to one boosts the current execution priority to 0.
- If AIRCR.PRIS is:

  **0**:

  Setting PRIMASK_NS to one boosts the current execution priority to 0.

  **1**:

  Setting PRIMASK_NS to one boosts the current execution priority to `0x80`.

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

**FAULTMASK**

In a PE without the Security Extension:

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

In a PE with the Security Extension, if AIRCR.BFHFNMINS is:

**0**:

Setting FAULTMASK_S to one boosts the current execution priority to -1.

If AIRCR.PRIS is:

**0**: Setting FAULTMASK_NS to one boosts the current execution priority to 0.

**1**: Setting FAULTMASK_NS to one boosts the current execution priority to `0x80`.

**1**:

Setting FAULTMASK_S to one boosts the current execution priority to -3.

Setting FAULTMASK_NS to one boosts the current execution priority to -1.

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

**BASEPRI**

In a PE without the Security Extension:

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

In a PE with the Security Extension:

- BASEPRI_S can be set to a priority number between 1 and the maximum supported priority number.
- If AIRCR.PRIS is:

  **0**: BASEPRI_NS can be set to a priority number between 1 and the maximum supported priority number.

  **1**: BASEPRI_NS can be set to a priority number between 1 and the maximum supported priority number. The value in BASEPRI_NS is then mapped to the bottom half of the priority range, so that the current execution priority is boosted to the mapped-to value in the bottom half of the priority range, that is from `0x80` to the supported maximum.

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, FAULTMASK and BASEPRI require M.*

$R_{FHMC}$ The PRIMASK, FAULTMASK, and BASEPRI priority boosting mechanisms only boost the group priority, not the subpriority.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, FAULTMASK and BASEPRI require M.*

$R_{SKBJ}$ Without the Security Extension:

- An exception return sets FAULTMASK to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S && M**.*

$R_{HRTM}$ With the Security Extension:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S && M**.*

$I_{LSXJ}$ The *raw execution priority* is:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{GBVL}$ The *requested execution priority* for a Security state is negative when any of the following are true:

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, FAULTMASK requires M.*

See also:

B3.13 *Priority model* on page 91.

B3.9 *Exception numbers and exception priority numbers* on page 80.

# B3.33 Lockup

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, entry to Debug state requites Halting debug.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, entry to Debug state requires Halting debug.*

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, entry to Debug state requires Halting debug.*

See also:

B3.13 *Priority model* on page 91.

Chapter B12 *Debug* on page 273.

## B3.33.1 Instruction-related lockup behavior

### Instruction execution

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, **FAULTMASK** requires M.*

$R_{QMMB}$ If the Security Extension is not implemented, a synchronous exception results in lockup when:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!S**.*

$R_{VGNW}$ Entry to lockup from an exception causes:

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

In addition, HFSR.FORCED is not changed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DWKP}$ Asynchronous BusFaults do not cause lockup.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KTQM}$ When a BusFault does not cause lockup, the value that is read or written to the location that generated the BusFault is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HTVD}$ ITSTATE does not advance when the PE is in lockup.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### Floating-point lazy Floating-point context preservation

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**. Note, an MPU violation requires MPU, an SAU violation requires S.*

$R_{MMBJ}$ When FPCCR.LSPEN is 0, any faults that are caused by floating-point register reads or writes during exception entry or exception return are handled as faults on stacking or unstacking respectively.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

### B3.33.2 Exception-related lockup behavior

#### Vector or stack pointer error on reset

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a Secure HardFault requires S.*

#### Errors on preemption and stacking for exception entry

$R_{VKTX}$ An AU violation, MPU violation, NOCP UsageFault, STKOF UsageFault, LSERR SecureFault, or synchronous BusFault during context stacking causes lockup when:

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an AU violation requires S, an MPU violation requires MPU, a UsageFault requires M, a SecureFault requires S.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an AU violation requires S, an MPU violation requires MPU, a UsageFault requires M, a SecureFault requires S, LSERR requires FP.*

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

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

At the point of lockup:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, an AU violation requires S, an MPU violation requires MPU, a UsageFault requires M, a SecureFault requires S, LSERR requires FP.*

### Vector read error on NMI or HardFault entry

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### Integrity checks on exception return

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{DFKP}$ When the PE enters lockup because of a fault that is generated by a failed integrity check, the PE:

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

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

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, CONTROL.FPCA and SFPA, FPSCR and S0-S31 require FP.*

$R_{XNZJ}$ When the PE enters lockup because of a fault that is generated by a failed integrity check, and MVE is implemented, VPR is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

## Errors when unstacking state on exception return

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{XFCQ}$ When a synchronous exception during context unstacking causes lockup, the PE:

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

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WZFG}$ When the PE enters lockup because of an AU violation, MPU violation, or synchronous BusFault during context unstacking, and MVE is implemented, VPR is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B3.22 *Exception return* on page 115.

# B3.34 Data independent timing

$I_{JFVR}$ The Armv8.1-M architecture supports Data independent timing operations.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**.*

$R_{DNPM}$ DIT behavior only applies if the instruction passes its Condition code check. The instruction remains subject to the rules of the architecture but is permitted to have a different execution time when compared to the same instruction that had passed the Condition code check.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**.*

$R_{NXXV}$ When AIRCR.DIT is set to 1, unless otherwise specified, the time required for Data independent timing operations is independent of all values that are accessed by operations from the following registers:

- FPCSR.{N,Z,C,V}.
- APSR.
- General-purpose registers.
- Floating-point Extension registers (S0-S31, D0-D15, and Q0-Q7).
- In a limited number of cases, VPR.P0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**. Note, Floating-point registers require FP or MVE, and VPR requires MVE.*

$R_{CWVH}$ When AIRCR.DIT is set to 1, this affects the following features:

- Exception handling. In addition to the standard set of registers, the following operations also exhibit Data independent timing for accesses to VPR.P0:
  - Exception entry.
  - Tail-chaining.
  - Lazy floating-point state preservation.
  - Exception return.
- EPSR.ICI. Whether a PE uses ICI for load/store multiple instructions is not dependent on the data values that are loaded or saved. This excludes the address that is being targeted.
- Beat wise execution. Whether a Data independent timing vector instruction overlaps with another vector instruction is not dependent on the data values being processed by the data independent timing vector instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**. Note, VPR and vector instructions requires MVE.*

$R_{WGVH}$ The DIT non-MVE instructions, including flag setting variants of the instruction are:

- Comparison and selection:
  - CMP (immediate), CMP (register), TEQ (immediate), TEQ (register), TST (immediate), TST (register).
  - CSEL, CSINC, CSINV, CSNEG.
- Arithmetic:
  - ADC (immediate), ADC (register), SBC (immediate), SBC (register).
  - ADD (SP plus immediate), ADD (SP plus register), ADD (immediate), ADD (immediate, to PC), ADD (register), SUB (SP minus immediate), SUB (SP minus register), SUB (immediate), SUB (immediate, from PC), SUB (register), RSB (immediate), RSB (register)
  - UMLAL, UMLAL.
- Bitwise:
  - AND (immediate), AND (register), BIC (immediate), BIC (register), EOR (immediate), EOR (register), MVN (immediate), MVN (register), ORN (immediate), ORN (register), ORR (immediate), ORR (register)
  - UBFX.
- Shifts, Bit Reversal:
  - ASR (immediate), ASR (register), LSL (immediate), LSL (register), LSL (immediate), LSR (register), ROR (immediate), ROR (register), RRX
  - RBIT, REV, REV16.
- Moves:
  - MOV (immediate), MOV (register).
  - MRS, MSR (register). Data independent timing is only required to be guaranteed for accesses to APSR.
- Load/store instructions. Data independent timing does not apply to the addresses that are being accessed, or to sign extending variants.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**.*

$R_{RFVV}$ The DIT MVE instructions are:

- Comparison and selection:
  - `VCMP (floating-point)`,`VCMP (vector)`, and `VPSEL`. `VPSEL` also exhibits Data independent timing with respect to the value of VPR.P0.
- Arithmetic:
  - `VADC`, `VSBC`, `VADD (vector)`, `VSUB (vector)`.
  - `VMULL (integer)` and `VMULL (polynomial)`.
- Bitwise:
  - `VAND`.
  - `VBIC (immediate)` and `VBIC (register)`.
  - `VEOR`.
  - `VMVN (immediate)` and `VMVN (register)`.
  - `VORN`.
  - `VORR`.
- Shifts, Bit Reversal:
  - `VBRSR`, `VSHR`, `VSHL`, `VSHLC`.
- Moves:
  - All vector VMOV instructions.
  - `VMSR`, `VMRS`. Data independent timing is only required to be guaranteed for accesses to FPSCR.{N,Z,C,V,QC} and VPR.P0.
  - `VREV16`, `VREV32`, `VREV64`.
- Load/store instructions. Data independent timing does not apply to the address that is being accessed, or to sign-extending variants.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT** && **MVE**.*

$R_{RWJW}$ For non-architected accesses, all instructions, including instructions that are not listed as Data independent timing instructions, exhibit Data independent timing with respect to data that is held in specified DIT registers that are not architecturally accessed by the instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DIT**.*

See also:

B5.4 *Beats* on page 170.

## B3.35 Context Synchronization Event

$R_{QXWD}$ The architecture requires a Context synchronization event to guarantee visibility of any change to any memory-mapped register described in the architecture. Following a Content synchronization event a completed write to a memory-mapped register is visible to an indirect read by an instruction appearing in program order after the context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TVHX}$ Between any change to a memory-mapped register and a subsequent Content synchronization event, it is UNPREDICTABLE whether an indirect read of the register by the PE uses the old or new values.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RMMM}$ Where multiple changes are made to memory-mapped registers before a Content synchronization event, each value might independently be the old or new value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NSLQ}$ Where unsynchronized values apply to different areas of architectural functionality, or IMPLEMENTATION DEFINED functionality, those areas might independently treat the values as being either the old or new value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BKSX}$ The choice between the behaviors is IMPLEMENTATION DEFINED and might vary for each use of the unsynchronized value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B3.36 Coprocessor support

$R_{BSLX}$ Coprocessor support is OPTIONAL.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{JBMG}$ When coprocessors are not supported, the fields in CPACR, NSACR, and CPPWR that are associated with the unsupported coprocessor are RAZ/WI.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{XSQH}$ The architecture supports 0-16 coprocessors, CP0 to CP15.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{HJDH}$ CP0 to CP7 are IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{XPRQ}$ It is IMPLEMENTATION DEFINED whether CP0 to CP7 can be used from both Secure and Non-secure states or whether the coprocessor is enabled for only Secure or Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**. Note, Secure state requires S.*

$R_{QSRC}$ Arm reserves CP8 to CP15.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{LKZM}$ CP10 to CP11 are reserved to support the Floating-point Extension, and CP10 controls the CP11 Floating-point instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{CSQD}$ From version 8.1-M of the architecture, access control for CP10 also controls CP8, CP9, CP14, and CP15.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$R_{LPMK}$ The state that is associated with Floating-point unit described in CPPWR.SU10 applies to S registers, D registers, and FPSCR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{VLNJ}$ From version 8.1-M of the architecture, the state that is associated with the Floating-point unit described in CPPWR.SU10 also applies to the Q registers and VPR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{XXDG}$ Instructions that are issued to unimplemented or disabled coprocessors result in a NOCP UsageFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{RMLV}$ If a coprocessor cannot complete an instruction, an UNDEFINSTR UsageFault is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

See also:

Chapter B4 *Floating-point Support* on page 150.

*CPACR, Coprocessor Access Control Register*

*CPPWR, Coprocessor Power Control Register*

# Chapter B4
# Floating-point Support

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

# B4.1 The optional Floating-point Extension, FPv5

$I_{VBNH}$ The optional Floating-point Extension defines a *Floating Point Unit* (FPU). Coprocessors 10 and 11 support the Extension.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{WPHK}$ The scalar Floating-point Extension can be implemented with or without MVE-F.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$I_{RXQX}$ Floating-point is sometimes abbreviated to FP.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{GQBM}$ The version of Floating-point Extension that is supported is FPv5.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{FGSG}$ FPv5 provides all of the following:

- Single-precision arithmetic operations.
- Optional double-precision arithmetic operations.
- Conversions between integer, double-precision, single-precision, and half-precision formats.
- Registers for Floating-point processing S0-S31, or D0-D15.
- Data transfers, between Arm general-purpose registers and FPv5 Extension registers S0-S31, or D0-D15, of single-precision and double-precision values.
- A Flush-to-zero mode that software can enable or disable.
- An optional *alternative half-precision* interpretation of the IEEE 754 half-precision encoding format.

FPv5 adds the following System registers:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{TVZF}$ From Armv8.1-M onwards, FPv5 provides Half-precision arithmetic operations.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$I_{PVBQ}$ When the Floating-point Extension is implemented, some software tools might require the following information:

| Extension | Single-precision arithmetic operations only | Single and double-precision arithmetic operations |
|---|---|---|
| FPv5 | FPv5-SP-D16-M | FPv5-D16-M |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{FTDS}$ When the Floating-point Extension is implemented, software can interrogate MVFR0, MVFR1, and MVFR2 to discover the Floating-point features that are implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{JDJQ}$ To use the Floating-point Extension, software must enable access to CP10, by writing to CPACR.CP10.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{PDMV}$ The value of CPACR.CP11 is UNKNOWN if it is not programmed to the same value as CPACR.CP10.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B7.1 *System address map* on page 241.

B4.2 *About the Floating-point Status and Control Registers* on page 153.

B4.3 *Registers for Floating-point data processing, S0-S31, or D0-D15* on page 154.

B4.8 *The Flush-to-zero mode* on page 160.

B4.9 *The Default NaN mode, and NaN handling* on page 162.

B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 157.

## B4.2 About the Floating-point Status and Control Registers

$I_{FQTM}$ For implementations of the Armv8.1-M architecture, FPCXT and VPR provide additional controls.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$R_{HCJS}$ The register map of the coprocessor System register space is as follows.

| Location | Register | Information |
|---|---|---|
| 0b0001 | FPSCR.{N,Z,C,V} | Access to flags |

All locations that are not explicitly listed in this table are reserved, and accesses to these locations result in UNPREDICTABLE behavior.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{KHDZ}$ The register map of the coprocessor System register space is as follows.

| Location | Register | Information |
|---|---|---|
| 0b0001 | FPSCR.{N,Z,C,V} | Access to flags |
| 0b0010 | FPSCR.{N,Z,C,V,QC} | Access to flags, including MVE saturation flag |
| 0b1100 | VPR | Privileged access to this register only |
| 0b1101 | VPR.P0 | Access to P0 field |
| 0b1110 | FPCXT_NS | Saves and restores the Non-secure FP context |
| 0b1111 | FPCXT_S | Saves and restores the Secure FP context |

All locations that are not explicitly listed in this table are reserved, and accesses to these locations result in UNPREDICTABLE behavior.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$I_{GJWP}$ Software can use VMRS and VMSR instructions to access the Floating-point Status and Control registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{DSSS}$ Software can use VMRS, VMSR `VLDR(System Register)`, and `VSTR(System Register)` instructions to access FPCXT, VPR, and the Floating-point Status and Control registers.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$R_{RRTZ}$ Accesses to the FPCXT will behave as `NOPs` unless both MVE and Floating-point extension are implemented.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** || **MVE**.*

$R_{WXZV}$ Accesses to the FPCXT are UNDEFINED from the Non-secure state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP** && **MVE** && **S**.*

$R_{FXBJ}$ Execution of Floating-point instructions that generate Floating-point exceptions update the appropriate status fields of FPSCR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B3.36 *Coprocessor support* on page 149.

B4.1 *The optional Floating-point Extension, FPv5* on page 151.

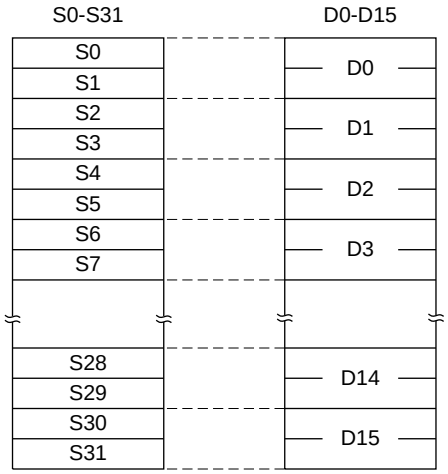
*FPSCR, Floating Point Status and Control Register*.

# B4.3 Registers for Floating-point data processing, S0-S31, or D0-D15

$R_{TWCB}$ The registers that FPv5 adds for Floating-point processing are visible as either:

- 32 single-precision registers, S0-S31.
- 16 double-precision registers, D0-D15.

These map as follows:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{XWJQ}$ After a Warm reset, the values of S0-S31 or D0-D15 are UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.1 *The optional Floating-point Extension, FPv5* on page 151.

B3.18 *Exception handling* on page 101.

## B4.4 Floating-point standards and terminology

$I_{XNMN}$ There are two editions of the IEEE 754 standard:

- IEEE 754-1985.
- IEEE 754-2008.

In this manual, references to IEEE 754 that do not include the year apply to either edition.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{MQFS}$ The Floating-point terminology that this manual uses differs from that used in IEEE 754-2008 as follows:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{BGPN}$ The following is called *Arm standard Floating-point operation*:

- IEEE 754-2008 plus the following configuration:
  - Flush-to-zero mode enabled.
  - Default NaN mode enabled.
  - Round to Nearest mode selected.
  - Alternative half-precision interpretation not selected.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

*IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.*

B4.8 *The Flush-to-zero mode* on page 160.

B4.9 *The Default NaN mode, and NaN handling* on page 162.

B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 157.

## B4.5 Floating-point data representable

$R_{FWXC}$ FPv5 supports the following, as defined by IEEE 754:

- Normalized numbers.
- Denormalized numbers.
- Zeros, +0 and -0.
- Infinities, $+\infty$ and $-\infty$.
- NaNs, signaling NaNsand quiet NaN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.4 *Floating-point standards and terminology* on page 155.

*IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008*.

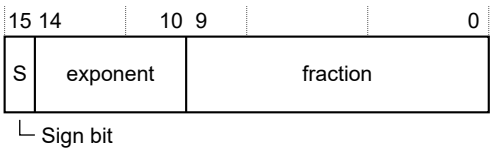
B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 157.

## B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision

$R_{RHKS}$ The half-precision, single-precision, and double-precision encoding formats are those defined by IEEE 754-2008, in addition to an alternative half-precision format.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{LGTJ}$ The half-precision encoding format is:

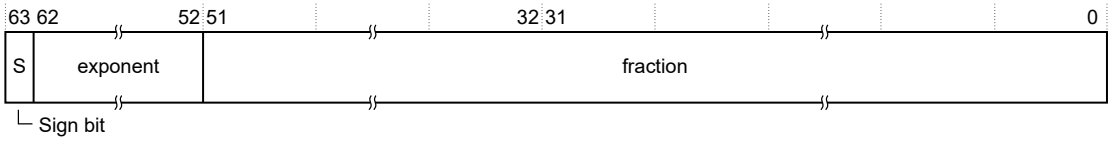



*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{CWBP}$ The single-precision encoding format is:


*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{FVWV}$ The double-precision encoding format is:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{RWRW}$ The interpretations of the half-precision, single-precision, and double-precision encoding formats are as follows.

**Half-precision**

There are two interpretations of the half-precision encoding formats:

- The interpretation that is defined by IEEE 754-2008.
- An alternative half-precision interpretation, indicated by FPSCR.AHP.

**Single-precision**

The interpretation that is defined by IEEE 754-2008.

**Double-precision**

The interpretation that is defined by IEEE 754-2008. See the following table:

| **E (biased exponent)** | **T (trailing significand)** | **S (sign bit)** | **T [51]** | **Value** |
|---|---|---|---|---|
| Zero for all formats. | Non-zero | - | - | A denormalized number. |
| - | Zero | 0 | - | Zero, +0 |
| - | - | 1 | - | Zero, -0 |
| Zero $<$ E $<$ `0x1F`, if one of the half precision formats. | - | - | - | A normalized number. |
| Zero $<$ E $<$ `0xFF`, if single-precision format. | - | - | - | - |
| Zero $<$ E $<$ `0x7FF`, if double-precision format. | - | - | - | - |
| `0x1F`, if half-precision format, IEEE interpretation. | Non-zero | - | 0 | A signaling NaN |
| `0xFF`, if single-precision format. | - | - | 1 | A quiet NaN |
| `0x7FF`, if double-precision format. | Zero | 0 | - | Infinity,$+\infty$ |
| - | Zero | 1 | - | Infinity,$-\infty$ |
| `0x1F`, if half-precision, alternative half-precision interpretation. | - | - | - | A normalized number. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{DPHH}$ The value of a normalized number is equal to:

**Half-precision**: $(-1)^S \times 2^{(E-15)} \times (1.T)$

**Single-precision**: $(-1)^S \times 2^{(E-127)} \times (1.T)$

**Double-precision**: $(-1)^S \times 2^{(E-1023)} \times (1.T)$

The value of a denormalized number is equal to:

**Half-precision**: $(-1)^S \times 2^{-14} \times (0.T)$

**Single-precision**: $(-1)^S \times 2^{-126} \times (0.T)$

**Double-precision**: $(-1)^S \times 2^{-1022} \times (0.T)$

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{PKXD}$ Denormalized numbers can be flushed to zero. FPv5 provides a Flush-to-zero mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

*IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.*

B4.5 *Floating-point data representable* on page 156.

## B4.7 The IEEE 754 Floating-point exceptions

$R_{BCCL}$ The IEEE 754 Floating-point exceptions are:

**Invalid Operation**: This exception is as IEEE 754-2008 (7.2) describes.

**Division by zero**: This exception is as IEEE 754-2008 (7.3) describes, with the following assumption:

- For the reciprocal and reciprocal square root estimate functions the dividend is assumed to be +1.0.

**Overflow**: This exception is as IEEE 754-2008 (7.4) describes.

**Underflow**: This exception is as IEEE 754-2008 (7.5) describes, with the additional clarification that:

- Assessing whether a result is tiny and non-zero is done before rounding.

**Inexact**: This exception is as IEEE 754-2008 (7.6) describes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{JCWS}$ The criteria for the Underflow exception to be generated are different in Flush-to-zero mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{NFHK}$ The corresponding status flags for the IEEE 754 Floating-point exceptions are FPSCR.{IOC, DZC, OFC, UFC, IXC}.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

*IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008.*

B4.8 *The Flush-to-zero mode* on page 160.

## B4.8 The Flush-to-zero mode

$I_{XGFP}$ Software can enable Flush-to-zero mode by setting FPSCR.FZ to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{WMKJ}$ Using Flush-to-zero mode is a deviation from IEEE 754.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{JQHX}$ Half-precision Floating-point numbers are exempt from Flush-to-zero mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{LVCG}$ In an Armv8.1-M implementation Half-precision Floating-point numbers are subject to Flush-to-zero mode.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$R_{VJSF}$ When Flush-to-zero mode is enabled, all single-precision denormalized inputs and double-precision denormalized inputs to Floating-point operations are treated as though they are zero, that is they are flushed to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{GGQW}$ In an Armv8.1-M implementation when Flush-to-zero mode is enabled, all half-precision denormalized inputs to Floating-point operations are treated as though they are zero, that is they are flushed to zero.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**.*

$R_{KBJJ}$ When an input to a Floating-point operation is flushed to zero, the PE generates an Input Denormal exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{SBCK}$ Input Denormal exceptions are only generated in Flush-to-zero mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{WJDM}$ When Flush-to-zero mode is enabled, the sequence of events for an input to a Floating-point operation is:

1. Flush to Zero processing takes place. If appropriate, the input is flushed to zero and the PE generates an Input Denormal exception.
2. Tests for the generation of any other Floating-point exceptions are done after Flush to Zero processing.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{PHPT}$ When Flush-to-zero mode is enabled, the result of a Floating-point operation is treated as if it is zero if, before rounding, it satisfies the condition:

$0 < \texttt{Abs(result)} <$ MinNorm, where:

- MinNorm is $2^{-126}$ for single-precision.
- MinNorm is $2^{-1022}$ for double-precision.

The result is said to be flushed to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{QPQF}$ When the result of a Floating-point operation is flushed to zero, the PE generates an Underflow exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{TPVD}$ In Flush-to-zero mode, the PE generates Underflow exceptions only when a result is flushed to zero. This uses different criteria than when Flush-to-zero mode is disabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{RTPH}$ When a Floating-point number is flushed to zero, the sign is preserved. That is, the sign bit of the zero matches the sign bit of the number being flushed to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **FP**.*

$R_{RWRT}$ The PE does not generate an Inexact exception when a Floating-point number is flushed to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{SQCJ}$ The corresponding status flag for the Input Denormal exception is FPSCR.IDC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

### B4.8.1 The Flush to zero mode half-precision calculations

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{MMKS}$ In an Armv8.1-M implementation Flush-to-zero mode mode is extended to include half-precision calculations.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**. Note, !8.0.*

$I_{NPCG}$ Software can enable Flush-to-zero mode for half-precision calculations by setting FPSCR.FZ16 to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**. Note, !8.0.*

$R_{KZFH}$ When Flush-to-zero mode is enabled, the result of a Floating-point operation is treated as if it is zero if, before rounding, it satisfies the condition:

$0 < \texttt{Abs(result)} <$ MinNorm, where:

- MinNorm is $2^{-14}$ for half-precision.
- MinNorm is $2^{-126}$ for single-precision.
- MinNorm is $2^{-1022}$ for double-precision.

The result is said to be flushed to zero.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**. Note, !8.0.*

$R_{HJZZ}$ The Effective value of FPSCR.FZ16 is zero when converting real values and integers from one Floating-point format to another.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**. Note, !8.0.*

$R_{LCDV}$ When Flush-to-zero mode is enabled for half-precision Floating-point and a half-precision Floating-point number is flushed to zero an Input Denormal Floating-point exception will not be generated.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FP**. Note, !8.0.*

See also:

B4.7 *The IEEE 754 Floating-point exceptions* on page 159.

## B4.9 The Default NaN mode, and NaN handling

$I_{FGPN}$ Software can enable Default NaN mode by setting FPSCR.DN to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{DJVH}$ Using Default NaN mode is a deviation from IEEE 754.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{QMQC}$ When Default NaN mode is enabled, the *Default NaN* is the result of both:

- All Floating-point operations that produce an untrapped Invalid Operation exception.
- All Floating-point operations whose inputs include at least one quiet NaN but no signaling NaNs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{NPRL}$ IEEE 754 specifies that:

- An operation that produces an untrapped Invalid Operation exception returns a quiet NaN as its result.

When Default NaN mode is disabled, behavior complies with this and adds:

- If the Invalid Operation exception was generated because one of the inputs to the operation was a signaling NaN, the quiet NaN result is equal to the first signaling NaN input with its most significant bit set to 1.
- The quiet NaN result is the Default NaN otherwise.

The *first signaling NaN input* means the first argument, in the left-to-right ordering of arguments, that is passed to the pseudocode function describing the operation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{VCSB}$ IEEE 754 specifies that:

- An operation using a Quiet NaN as an input, but no signaling NaNs as inputs, returns one of its quiet NaN inputs as its result.

When Default NaN mode is disabled, behavior complies with this and adds:

- The Quiet NaN result is the first Quiet NaN input.

The *first quiet NaN input* means the first argument, in the left-to-right ordering of arguments, that is passed to the pseudocode function describing the operation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$I_{LXLF}$ Depending on the Floating-point operation, the exact value of a Quiet NaN result might differ in both sign and the number of T bits from its source.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.10 *The Default NaN* on page 163.

B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 157.

## B4.10 The Default NaN

$R_{FQFG}$ The Default NaN is:

| Field | Half-precision, IEEE 754-2008 interpretation | Single-precision | Double-precision |
|---|---|---|---|
| S | 0 | 0 | 0 |
| E | `0x1F` | `0xFF` | `0x7FF` |
| T | bit[9] == 1, bits[8:0] == 0 | bit[22] == 1, bits[21:0] == 0 | bit[51] == 1, bits[50:0] == 0 |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.6 *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page 157.

B4.9 *The Default NaN mode, and NaN handling* on page 162.

## B4.11 Combinations of Floating-point exceptions

$I_{BTTH}$ In compliance with IEEE 754:

- An Inexact Floating-point exception can occur with an Overflow Floating-point exception.
- An Inexact Floating-point exception can occur with an Underflow Floating-point exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

$R_{LFVH}$ An Input Denormal exception can occur with other Floating-point exceptions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.7 *The IEEE 754 Floating-point exceptions* on page 159.

B4.8 *The Flush-to-zero mode* on page 160.

## B4.12 Priority of Floating-point exceptions relative to other Floating-point exceptions

$R_{PLHJ}$ Some Floating-point instructions specify more than one Floating-point operation. In these cases, an exception on one operation is higher priority than an exception on another operation when generation of the second exception depends on the result of the first operation. Otherwise, it is UNPREDICTABLE which exception is higher priority.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FP**.*

See also:

B4.7 *The IEEE 754 Floating-point exceptions* on page 159.

# Chapter B5
# Vector Extension

This chapter specifies the optional Armv8.1-M Vector Extension rules. It contains the following sections:



*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

# B5.1 Vector Extension operation

$I_{LRKM}$ MVE-I operates on 32-bit, 16-bit, and 8-bit data types, including Q7, Q15, Q31 integer values. MVE-F operates on half-precision and single-precision floating-point values.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{JYRS}$ Vector instructions operate on a fixed vector width of 128 bits.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{MSHF}$ Integer MVE instructions can be implemented with or without the scalar Floating-point Extension.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE-I**.*

$I_{LXMG}$ An implementation that includes MVE also includes the DSP Extension.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{QRWY}$ Vector operations are divided in two orthogonal ways:

- Lanes.
- Beats.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{FNYX}$ The word Element is used in this specification to refer to the data that is put into a lane.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{VXBF}$ Multiple lanes can be executed per beat. There are four beats per vector instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{BDHN}$ The pseudocode for each vector instruction is executed four times, one time for each beat. The `GetCurInstrBeat()` function returns the current beat number and predication details. These determine which of the lanes are operated on during the current execution of the code.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{LFDY}$ Multiple Element writes that are generated by the same vector store instruction by the same observer can be observed in any order, with the exception that writes to the same location by different Elements are observed in order of increasing vector element number.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{PLFG}$ Some instructions permit the use of a *zero register* (ZR) as a scalar source operand, as indicated in the individual instruction descriptions. ZR is encoded as the value `0b1111` when a 4-bit register specifier is used. ZR is RAZ/WI.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B3.28 *Low overhead loops* on page 129.

B3.18 *Exception handling* on page 101.

B5.2 *Vector register file* on page 168.

B5.3 *Lanes* on page 169.

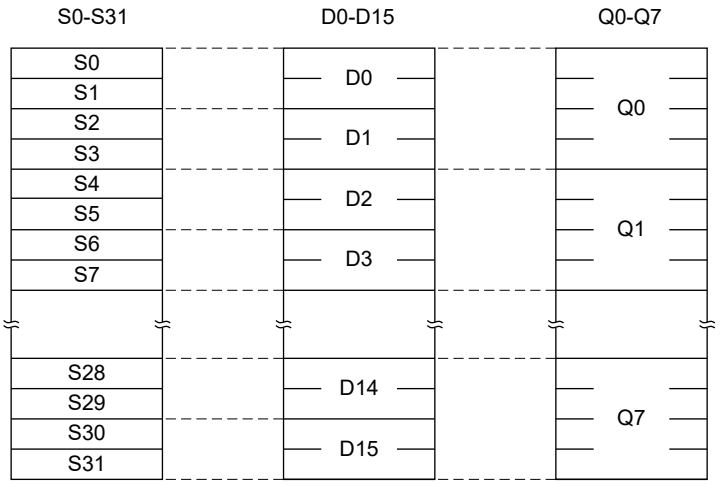
B5.4 *Beats* on page 170.

## B5.2 Vector register file

$R_{BBYH}$ MVE defines eight vector registers that alias onto the Floating-point Extension register file.

```
Q[0]<127:96> = S3, Q[0]<95:64> = S2, Q[0]<63:32> = S1, Q[0]<31:0> = S0
Q[1]<127:96> = S7, Q[1]<95:64> = S6, Q[1]<63:32> = S5, Q[1]<31:0> = S4
...
Q[7]<127:96> = S31, Q[7]<95:64> = S30, Q[7]<63:32> = S29, Q[7]<31:0> = S28
```

These registers map as follows:




*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***MVE****.*

$R_{PPPV}$ If CP10 is enabled, access to vector register 0-7 is permitted, unless otherwise stated in the individual instruction descriptions.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***MVE****.*

$I_{MFLD}$ To reduce pressure on the vector register file, many vector instructions can use scalar arguments from the general-purpose register file.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards.*

$I_{WWPZ}$ After a Warm reset, the values of Q0-Q7 are UNKNOWN.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards.*

See also:

C1.4 *Instruction set encoding information* on page 435.

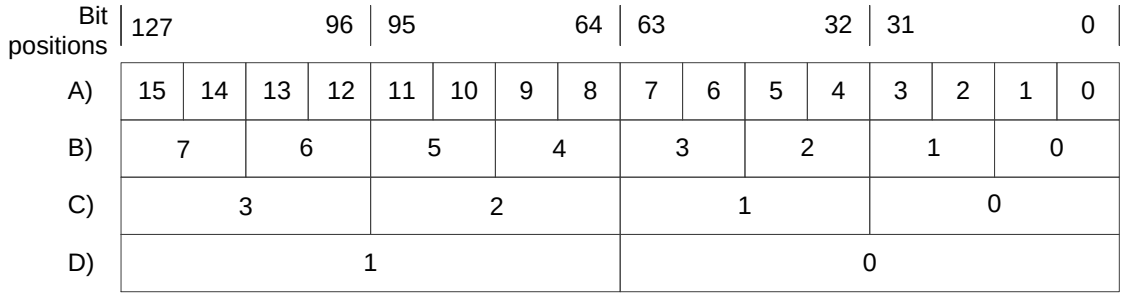
# B5.3 Lanes

$R_{DWVD}$ The lane width of the operation to be performed is specified by the instruction that is being executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{QSBC}$ The permitted lane widths, and lane operations per beat, are:

- For a 64-bit lane size, a beat performs half of the lane operation.
- For a 32-bit lane size, a beat performs a one lane operation.
- For a 16-bit lane size, a beat performs a two lane operations.
- For an 8-bit lane size, a beat performs a four lane operations.




A) 8-bit lane numbers
B) 16-bit lane numbers
C) 32-bit lane numbers
D) 64-bit lane numbers

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

Chapter C2, Instruction specification

# B5.4 Beats

$I_{YYZD}$ A vector instruction executes beats sequentially, from beat 0-3.

| Bit position | 127 96 | 95 64 | 63 32 | 31 0 |
|---|---|---|---|---|
| Beat number | 3 | 2 | 1 | 0 |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{PCBB}$ The number of beats for each tick describes how much of the architectural state is updated for each Architecture tick in the common case. In a trivial implementation, an Architecture tick might be one clock cycle:

- In a single-beat system, one beat might occur for each tick.
- In a dual-beat system, two beats might occur for each tick.
- In a quad-beat system, four beats might complete for each tick.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{XTZH}$ It is IMPLEMENTATION DEFINED how many beats are executed for each Architecture tick.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{MWSJ}$ The number of beats per tick might change at runtime and is not required to be constant.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{JSHB}$ Multiple faults might occur within a single Architecture tick. In this case, only one fault is raised. The fault that is generated is determined using the following priorities:

- The fault from the oldest instruction takes priority.
- If multiple faults are associated with the oldest faulting instruction, the fault that was generated by the lowest numbered Element takes priority.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{LHXJ}$ An exception can be taken on any beat of a vector instruction. RETPSR.ECI in the exception stack frame stores information about how many beats of the instruction at the return address and how many beats of the subsequent instruction have been executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{ZBKX}$ A dual-beat overlap system implies that the last two beats of a vector instruction can overlap with the first two beats of the next vector instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{GNLS}$ The following is an example of a dual-beat system where two beats are executed per Architecture tick. The figure labels are:

**Tick** Architecture tick.

**A0-A3** Beats of the `VLDRW` instruction.

**B0-B3** Beats of the `VMUL` instruction.

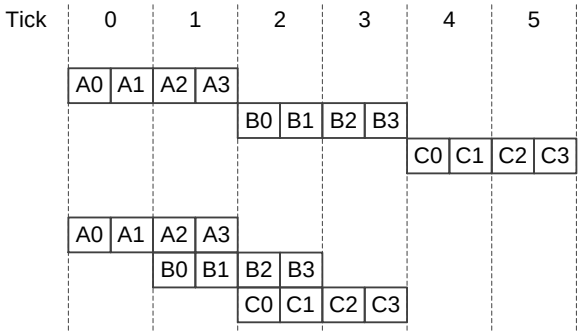
**C0-C3** Beats of the `VSHR` instruction.

Vector instructions not overlapping

```
VLDRW.U32 Q1, [R0],#16
VMUL.I32  Q0, Q1,  Q2
VSHR.U32  Q0, Q0,  #1
```

Vector instructions overlapping

```
VLDRW.U32 Q1, [R0],#16
VMUL.I32  Q0, Q1,  Q2
VSHR.U32  Q0, Q0,  #1
```




EPSR.ECI explains how beats are captured in the ECI field.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{VWBD}$ The PE can restart execution from any valid ECI value, even if the PE cannot generate all the ECI values.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{KRNF}$ Instructions that are subject to beatwise execution can only overlap if they are consecutive in the execution order.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{FKCG}$ The architecturally visible overlap of instructions is only permitted for instructions subject to beatwise execution if:

- The overlap does not violate data dependencies between instruction beats.
- The overlap is not between two instructions subject to beatwise execution that both access memory.
- In a low overhead loop, the overlap does not violate LR hazard.
- The overlap is not between an instruction before a BF branch point and the instruction at the target of the BF.
- An implicit `LE, LETP` instruction is executed at the end of a loop body when LO_BRANCH_INFO is valid and the instruction after the implicit `LE, LETP` instruction in execution order is subject to beatwise execution.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{NQKF}$ Vector instructions are permitted to overlap if the data dependency is at beat granularity and not at instruction granularity.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{PRSG}$ After each Architecture tick, the architectural instruction overlap is representable by a valid EPSR.ECI value.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B5.6.1 *Loop tail predication* on page 176.

B3.29 *Branch future* on page 132.

B5.5 *Exception state* on page 172.

B3.28 *Low overhead loops* on page 129.

Chapter C2, Instruction specification

## B5.5 Exception state

$R_{GFXK}$ The architecture supports taking exceptions in the middle of multiple partially executed instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{TTZB}$ LR is updated when LOB handling causes the PC to return to the start of the loop. The PC is only updated when all beats of an instruction have completed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{NTYR}$ For exceptions that occur in the middle of a beat-wise vector instruction that is executing:

- The exception return address points to the oldest incomplete instruction.
- RETPSR.ECI in the exception stack frame stores information about how many beats of the instruction at the return address, and how many beats of the subsequent instruction, have already been executed.

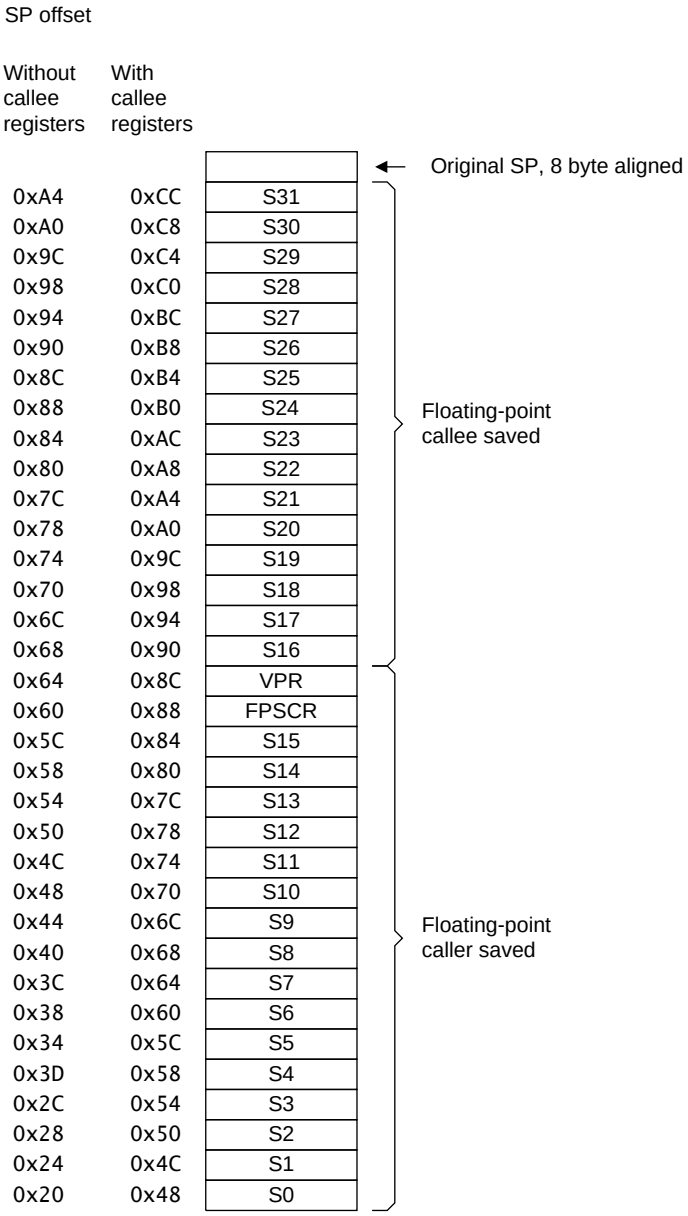
*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{THNH}$ When returning from an exception, valid RETPSR.ECI values indicate the completed instruction beats.

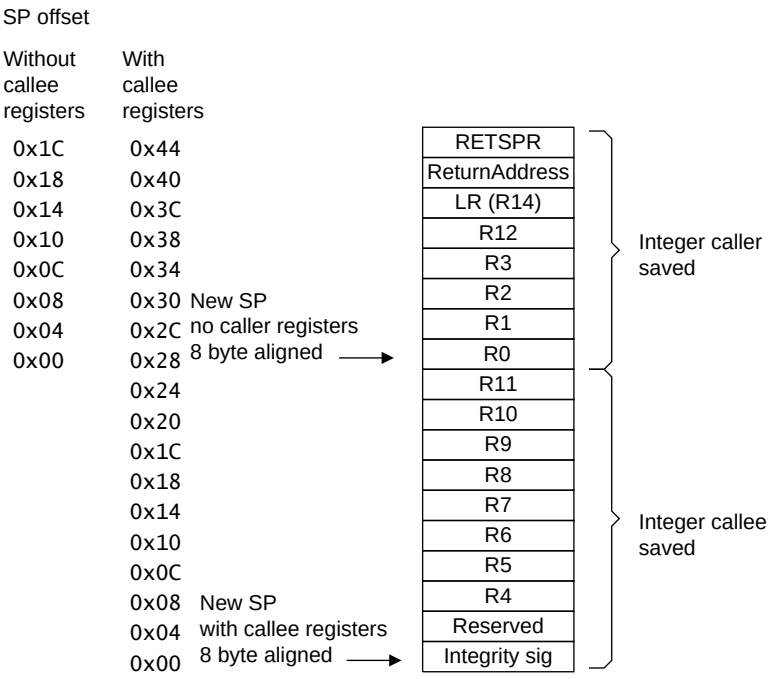
*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{XXRL}$ The existing exception stack frame format is modified to store the VPR register in the previously reserved location above FPSCR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*




*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards.*




*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{ZPTT}$ In EPSR, XPSR, and RETPSR, the ECI and ICI fields, and ITSTATE overlap.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{KFZJ}$ The PE does not generate an INVSTATE Usage fault if a nonzero value in EPSR.ICI corresponds to a valid value in EPSR.ECI, and the instruction that is being executed is:

- A vector instruction that is subject to beat-wise execution.
- An `LE, LETP` instruction.
- An FPB generated breakpoint or a `BKPT` instruction.

The execution of the breakpoint or LE instruction does not advance any of the register fields that are used for instruction beat execution tracking.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**. Note, FPB requires FPB.*

$I_{GRFW}$ The architecture tracks the completion of beats within vector instructions. Because the Element size can be smaller than the beat size, it is possible that an exception might be generated for a beat that has only partially completed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{SXTM}$ If execution of a beat is abandoned, then:

- RETPSR.ECI only indicates that a beat is completed if all the Elements that are associated with the beat have been completed.
- If the destination register is not the same as the source register for an abandoned instruction, the parts of a vector destination register that are associated with an abandoned beat, and all subsequent beats of the abandoned instruction, are set to an UNKNOWN value.
- Any scalar destination registers, the VPR state, and the FPSCR.QC flag record all the architecture state updates that are associated with the fully completed beats. Updates that are associated with the abandoned beat and all subsequent beats of the instruction are not recorded.

Partial stores to locations that might be accessed by the abandoned beat and all subsequent beats might be observed. Loads to locations of the abandoned beats and all subsequent beats might be observed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{DDNC}$ The return address for the instruction fetch fault, an UNDEFINSTR Fault, or a NOCP Usage fault is always the address of the instruction that triggered the fault. The fault is taken after all the preceding instructions have completed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{ZLFY}$ If an exception is taken during the execution of overlapping beat-wise executable instructions, this might become architecturally visible.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{KDNH}$ Architecture state updates that are associated with an Architecture tick are observed as one of the following:

- All updates to the architecture state are observable.
- Partial updates to the architecture states (both to the registers and to memory) are permitted for instructions that can be restarted without data corruption.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

## B5.6 Predication/conditional execution

$R_{RSLP}$ MVE includes predication that enables the independent masking of each lane within a vector operation. It supports the following predication mechanisms:

- Loop tail predication. This eliminates the requirement for special vector tail handling code after loops where the number of Elements to be processed is not a multiple of the number of Elements in the vector.
- VPT predication. This enables data-dependent conditions that are based on data value comparisons to mask each vector lane separately.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{QGLY}$ Loop tail predication and VPT predication operate separately. The resulting predication flags from each mechanism are ANDed together so that a lane of a vector operation is only active if both the loop tail predication and the VPT predication conditions are true.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B5.6.1 *Loop tail predication* .

B5.6.2 *VPT predication* on page 177.

B5.6.3 *Effects of predication* on page 180.

### B5.6.1 Loop tail predication

$R_{DCZN}$ Low overhead loops can be used with vector instructions, for example with a word-based memory copy instruction. The number of words to copy might not be a multiple of the vector length, therefore loop tail predication can eliminate any additional tail handling steps.

MVE includes special loop tail predication instructions, `WLSTP`, `DLSTP`, `LETP`, and `LCTP`, that operate as follows:

- The source register of the loop start instruction contains the number of vector Elements that are to be processed, instead of the iteration count.
- The loop start instruction sets FPSCR.LTPSIZE to the requested Element size. This alters the amount by which the Element count in LR is decremented at the end of each loop iteration.
- On the last iteration of the loop, the values in LR and FPSCR.LTPSIZE determines the number of vector lanes that are to be masked.
- After the last instruction of the last loop iteration has been executed, tail predication is disabled by setting FPSCR.LTPSIZE to `0b100`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{RBPP}$ The active floating-point state is defined by `ActiveFPState()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{GQSH}$ To prevent the inadvertent creation of floating-point contexts and the predication of vector operation outside of a loop, FPSCR.LTPSIZE behaves as follows:

- FPSCR.LTPSIZE reads as `0b100` if there is no active floating-point state.
- FPSCR.LTPSIZE is set to `0b100` if any of the following events occur:
  - On the last iteration of a loop by either the execution of an `LETP` instruction, or by execution reaching the end of the loop body when LO_BRANCH_INFO is valid and the floating-point context is active.
  - An `LCTP` instruction is executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{NRTL}$ Arm recommends that tail predicated loop start instructions are only used with a tail predicated loop end instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{BDJB}$ FPDSCR.LTPSIZE always reads as `0b100`, and therefore the floating-point contexts that are automatically initialized are created with predication disabled.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

### B5.6.2 VPT predication

$R_{ZFLV}$ Comparison-based predication is supported by vector predication blocks.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{CVTC}$ A vector predication block is called a VPT block. A VPT block is defined as the *n* instructions following a `VPT` or `VPST` instruction, where *n* is the number of instructions that the `VPT` or `VPST` instruction defines as being subject to predication conditions. The predication conditions are stored in the VPR register. *n* is less than or equal to 4.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{BVPG}$ The instructions in a VPT block can be subject to either the condition or to the inverse of the condition.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{RVJJ}$ A `VCMP (vector)` or a `VCMP (floating-point)` instruction can be placed inside a VPT block. `VCMP` instructions update the predication flag on completion, therefore affecting the subsequent instructions in the VPT block. The subsequent instructions in the VPT block are subject to the predicates of the VPT block and the updates caused by the `VCMP` instructions. The execution of successive `VCMP` instructions permits the creation of complex predication conditions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$I_{PHJP}$ Allowing instructions to be subject to either the condition or the inverse of the condition enables the instructions in both the THEN (T decorator) and the ELSE (E decorator) parts of an IF statement to be predicated with a single VPT instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{VWGT}$ The following table shows the VPT instruction variants, mask field encodings, and the associated decorators that are placed on the subsequent instructions.

| Instruction name | Mask value | Number of subsequent instructions to be predicated | <v> instruction decorator | | | |
|---|---|---|---|---|---|---|
| | | | First | Second | Third | Fourth |
| VPT | 0b1000 | 1 | T | - | - | - |
| VPTT | 0b0100 | 2 | T | T | - | - |
| VPTE | 0b1100 | 2 | T | E | - | - |
| VPTTT | 0b0010 | 3 | T | T | T | - |
| VPTTE | 0b0110 | 3 | T | T | E | - |
| VPTEE | 0b1010 | 3 | T | E | E | - |
| VPTET | 0b1110 | 3 | T | E | T | - |
| VPTTTT | 0b0001 | 4 | T | T | T | T |
| VPTTTE | 0b0011 | 4 | T | T | T | E |
| VPTTEE | 0b0101 | 4 | T | T | E | E |
| VPTTET | 0b0111 | 4 | T | T | E | T |
| VPTEEE | 0b1001 | 4 | T | E | E | E |
| VPTEET | 0b1011 | 4 | T | E | E | T |
| VPTETT | 0b1101 | 4 | T | E | T | T |
| VPTETE | 0b1111 | 4 | T | E | T | E |

The same encoding format is used for VPST.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{LKXQ}$ VPR contains a MASK field for each pair of beats of a vector instruction. This permits beat-wise overlapping of the VPT or VPST instructions with the surrounding vector instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{KHCV}$ The state of VPR is UNKNOWN when use of a VPT block results in CONSTRAINED UNPREDICTABLE behavior.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{XHQR}$ The following conditions result in CONSTRAINED UNPREDICTABLE behavior when they apply to a VPT block:

- The presence of a non-VPT compatible instruction in a VPT block. This includes:
  - All instructions that are not part of MVE, with the exception of BKPT.
  - MVE instructions that are marked as not being VPT compatible.
- A BF branch point within a VPT block.
- Branching into a VPT block.
- Exception return or returns from Debug state if VPR.{MASK23, MASK01} is not consistent with the position returned to in the VPT block.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**. Note, Debug state requires Halting Debug.*

$R_{DZWZ}$ The CONSTRAINED UNPREDICTABLE behavior for a VPT block is one of the following:

- The VPT or VPST instruction generates an UNDEFINED Instruction fault.
- The instruction that causes the CONSTRAINED UNPREDICTABLE behavior does one of the following:
  - It raises an UNDEFINED Instruction fault.
  - It executes normally.
  - It has UNKNOWN predication applied.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{XXKG}$ The CONSTRAINED UNPREDICTABLE behavior for a VPT compatible instruction executed outside a VPT block when the VPR mask is non-zero is one of the following:

- It raises an UNDEFINED Instruction fault.
- It executes normally.
- It has UNKNOWN predication applied.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{LQDR}$ If VPR.{MASK32, MASK01} is nonzero, the execution of a non-VPT compatible instruction outside of a VPT block is not UNPREDICTABLE and does not advance VPT state. The VPR state is only advanced after the completion of a pair of beats within a vector instruction that is subject to beat-wise execution.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{XMHZ}$ In the case of an exception return or a return from Debug state, the instruction that exhibits the CONSTRAINED UNPREDICTABLE behavior is defined as the instruction that is being returned to.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**. Note, Debug state requires Halting debug.*

$R_{MBQX}$ For a BF branch point within a VPT block, the instruction that exhibits the CONSTRAINED UNPREDICTABLE behavior can be one of the following:

- The instruction before the BF branch point.
- The instruction after the BF branch point.
- The instruction at the BF branch target address.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{SZQK}$ For the purposes of the CONSTRAINED UNPREDICTABLE behavior described in this section, a memory location is considered to be in VPT block until:

- The VPT or VPST instruction has been removed.
- All the addresses that are covered by the VPT block have been invalidated in the instruction cache (if implemented).
- A subsequent Context synchronization event has occurred.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{KKNH}$ There are similarities between VPT/VPR and IT/ITSTATE, but there are also some important differences as follows:

- Unlike IT, the VPT instruction performs the actual comparison in addition to applying the result to the subsequent instructions. As such, VPT can be considered as the vectorized combination of CMP and IT.
- There are multiple MASK fields in VPR that handle partial instruction execution caused by exceptions during the overlapping of instructions.
- The MASK fields are similar to ITSTATE[3:0] and encode both the number of instructions outstanding in the current VPT block, and whether these instructions are subject to the THEN or the ELSE condition.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{HJGT}$ VPR.P0 contains one predication bit per 8-bit lane. The VPR mask bits cause the VPR predication bits to be inverted if the corresponding mask bit is set to 1. The mask bits that are shifted out toggle the current predication condition and are not part of the predication condition. The value of VPR.MASK01 affects bits[7:0] of VPR.P0 and the value of VPR.MASK23 affects bits [15:8] of VPR.P0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{PXMY}$ The VPR predication bits are not inverted after executing the last instruction in a VPT block.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{LRVJ}$ The state in the VPR register can be accessed directly using `VMRS`, `VMSR`, `VLDR (System Register)`, and `VSTR (System Register)` instructions. Setting VPR using a `VMSR` or `VLDR (System Register)` instruction does not make the instructions that follow `VMSR` or `VLDR (System Register)` part of a VPT block.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{MPTV}$ Execution of a VPT compatible instruction outside of a VPT block with a nonzero value in VPR.{MASK23, MASK01} results in CONSTRAINED UNPREDICTABLE behavior and does one of the following:

- It raises an UNDEFINED Instruction fault.
- It executes normally.
- It has UNKNOWN predication applied.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

Chapter C1 *Instruction Set Overview* on page 420.

### B5.6.3 Effects of predication

$I_{TZPD}$ The exact effects of a false predication value are defined in the instruction pseudocode.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{SYDC}$ Vector predication has no effect on scalar instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{ZLSJ}$ For non-load instructions for vector register file writes, predication is always performed at byte level granularity, regardless of the Element size that is specified by the vector instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{VHSN}$ For non-load instructions, the predicate flags determine if the destination register byte is updated with the new value or if the previous value is preserved. For load instructions, where lanes are predicated false, the corresponding parts of the destination register are set to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{LYPK}$ For base pointer write-back, vector predication does not affect address write-back in load and store instructions. This applies both when the address is in a scalar register, and when it is in a vector register.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{LNDR}$ The predication flag determines whether a lane operation is performed. For Element sizes of more than 8 bits for the types of instruction listed here, the LSB of the corresponding group of predicate flags determines:

- For vector operations that perform reduction across the vector and produce a scalar result, whether the value is accumulated or not.
- For vector store instructions, whether the store occurs or not.
- For vector load instructions, whether the value that is loaded or whether zeros are written to that element of the destination register.
- The setting of the FPSCR.QC saturation flags.

For predication, 64-bit vector memory load/store operations are treated as if they were a pair of 32-bit operations.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{CJRK}$ The relation between lane width and bits in VPR.P0 is as follows:

| Lane width | Bits in VPR.P0 |
|---|---|
| 32 bits | [12, 8, 4, 0] |
| 16 bits | [14, 12, 8, 6, 4, 2, 0] |
| 8 bits | [15:0] |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

See also:

B5.6.1 *Loop tail predication* on page 176.

B5.6.2 *VPT predication* on page 177.

### B5.6.4 IT block

$R_{PZDX}$ Instructions that are subject to beat-wise execution are not permitted in IT blocks. For the exceptions to this rule, see the decode pseudocode in the individual instruction descriptions. In these exceptional cases, beat-wise execution is not performed and the instruction does not overlap with other instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

## B5.7 MVE interleaving/de-interleaving loads and stores

$I_{VWWN}$ For implementations that include MVE, data streams can be interleaved and de-interleaved with strides of 2 and 4, using `VLD2`/`VLD4` and `VST2`/`VST4`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{CMZP}$ The interleaving and de-interleaving instructions always operate on 128 bits of data at a time.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{XGTN}$ When using `VLD4`, each of the four instructions loads 128 bits of data, and partially updates the four destination vector registers. The memory offsets and destination register sections that are accessed are arranged so that when all four instructions have been executed, the de-interleaving operation has been performed.

VLD40.32 {Q0-Q3}, [Rn]
VLD41.32 {Q0-Q3}, [Rn]
VLD42.32 {Q0-Q3}, [Rn]
VLD43.32 {Q0-Q3}, [Rn]!

| | | | | |
|---|---|---|---|---|
| Q3 | S15=Mem[60] | S14=Mem[44] | S13=Mem[28] | S12=Mem[12] |
| Q2 | S11=Mem[56] | S10=Mem[40] | S9=Mem[24] | S8=Mem[8] |
| Q1 | S7=Mem[52] | S6=Mem[36] | S5=Mem[20] | S4=Mem[4] |
| Q0 | S3=Mem[48] | S2=Mem[32] | S1=Mem[16] | S0=Mem[0] |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{NSFK}$ The assembly syntax for `VLD2`/`VLD4` and `VST2`/`VST4` lists the range of vector registers to be accessed. Only the lowest numbered register is encoded in the opcode. If this register number plus the number of registers to be accessed is greater than 7 (the highest numbered vector register) behavior is a CONSTRAINED UNPREDICTABLE choice of the following:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.
- One or more of the vector registers become UNKNOWN. If the instruction specifies write-back, the base register becomes UNKNOWN. No other general-purpose registers are affected.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

# Chapter B6
# **Memory Model**

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

# B6.1 Memory accesses

$I_{XRDS}$ The memory accesses that are referred to in describing the memory model are instruction fetches from memory and load or store data accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LKQN}$ The instruction operation uses the `MemA()` or `MemU()` helper functions. If the Main Extension is not implemented unaligned accesses using the `MemU()` helper functions generate an alignment fault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BFNF}$ A memory access is governed by:

- Whether the access is a read or a write.
- The address alignment.
- Data endianness.
- Memory attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FLFQ}$ Memory reads that are generated by MVE instructions using `MemA_MVE()` are allowed to access bytes that are not explicitly accessed by the instruction if the bytes that are accessed are in a 32-byte window that is aligned to 32 bytes, and if that window contains at least one byte that is explicitly accessed by the instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{SKLB}$ Arm recommends that software does not use vector load/store instructions with data in volatile memory.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{LMBL}$ If an MVE load or store operation results in an access to the Private Peripheral Bus (PPB) address space, within the System region of the system address map, the behavior of the accesses is CONSTRAINED UNPREDICTABLE and is one of the following:

- It generates a Bus Fault.
- The specified access to the PPB address space is performed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B6.11 *Ordering of implicit memory accesses* on page 201.

B6.12 *Ordering of explicit memory accesses* on page 202.

B6.14 *Normal memory* on page 208.

B6.16 *Device memory* on page 211.

B6.20 *Memory access restrictions* on page 219.

B7.2 *The System region of the system address map* on page 242.

## B6.2 Address space

$R_{FFMK}$ The address space is a single, flat address space of $2^{32}$ bytes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SNPV}$ In the address space, byte addresses are unsigned numbers in the range $0\text{-}2^{32}\text{-}1$.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RGBT}$ If an address calculation overflows or underflows the address space, it wraps around. Address calculations are modulo $2^{32}$.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JTKM}$ Normal sequential execution cannot overflow the top of the address space, because the top of memory always has the Execute Never (XN) memory attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BPMP}$ One or more accesses that target or wrap around the top or bottom bytes of memory, access a sequence of words at increasing memory addresses, effectively incrementing the address by four for each load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, The encodings of some instructions require M, the encodings of some instructions require FP.*

$R_{ZXDN}$ Where an exception entry or tail-chaining accesses bytes on the stack that span the top or bottom of the 32-bit memory address space, it is IMPLEMENTATION DEFINED whether stack limit checking is applied.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

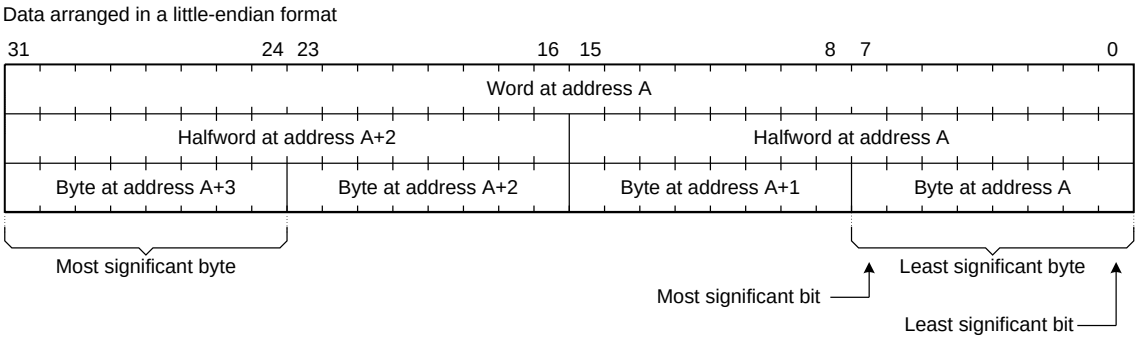
Chapter B7 *The System Address Map* on page 240.

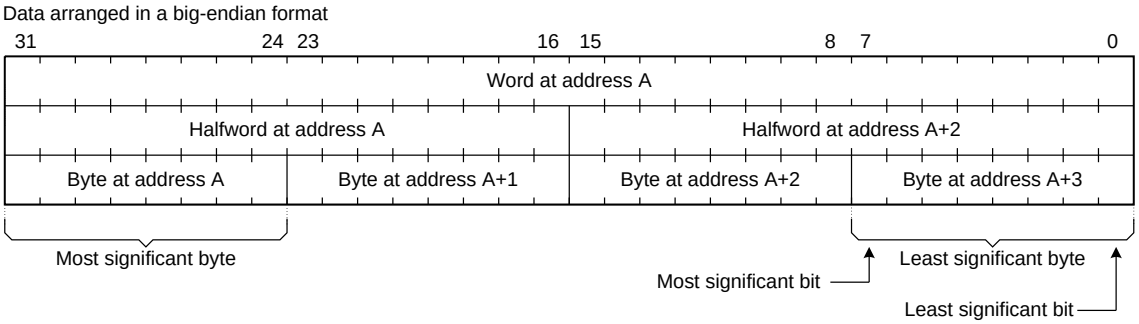
# B6.3 Endianness

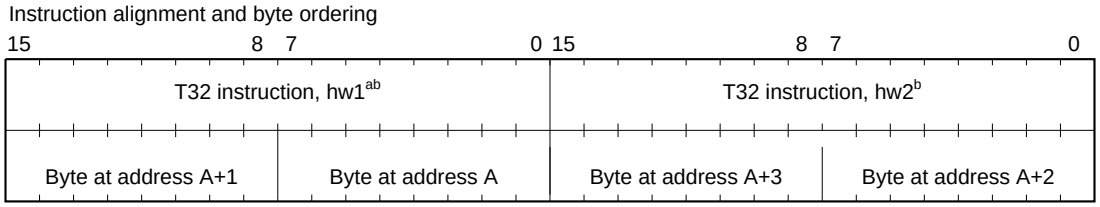
$I_{CTVV}$ In memory:

The following figures show the relationship between:

- The word at address A.
- The halfwords at addresses A and A+2.
- The bytes at addresses A, A+1, A+2, and A+3.




a) Bits[15:0]: this is hw 1 for a T32 instruction with a 16-bit encoding

b) Bits[31:0]: this is hw1 and hw2 for a T32 instruction with a 32-bit encoding

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JJQL}$ Instruction fetches are always little-endian, which means that the PE assumes a little-endian arrangement of instructions in memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MNSB}$ All accesses to the Private Peripheral Bus (PPB) are always little-endian, which means that the PE assumes a little-endian arrangement of the PPB registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TFKG}$ The endianness of data accesses is IMPLEMENTATION DEFINED, as indicated by AIRCR.ENDIANNESS.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KPCF}$ AIRCR.ENDIANNESS is either:

- Implemented with a static value.
- Configured by a hardware input on reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XDJV}$ Instructions that cause a memory access that crosses the PPB boundary are CONSTRAINED UNPREDICTABLE if AIRCR.ENDIANNESS is set to 1. The permitted behavior is one of the following:

- The instruction behaves as a NOP.
- The instruction raises an UNALIGNED UsageFault.
- If the instruction that crossed the PPB boundary was a load, the value of the destination register becomes UNKNOWN.
- If the instruction that crossed the PPB boundary was a store, the value of the memory locations accessed becomes UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a UsageFault requires M.*

$R_{QHWC}$ For data accesses, the following table shows the data element size that endianness applies to, for endianness conversion purposes.

| Instruction class | Instructions | Element size |
|---|---|---|
| Load or store byte | LDR{S}B{T}, LDAB, LDAEXB, STLB, STLEXB, STRB{T}, TBB, LDREXB, STREXB | Byte |
| Load or store halfword | LDR{S}H{T}, LDAH, LDAEXH, STLH, STLEXH, and STRH{T}, TBH, LDREXH, STREXH | Halfword |
| Load or store word | LDR{T}, LDA, LDAEX, STL, STLEX, and STR{T}, LDREX, STREX, VLDR.F32, VSTR.F32 | Word |
| Load or store two words | LDRD, STRD, VLDR.F64, VSTR.F64 | Word |
| Load or store multiple words | LDM{IA, DB}, STM{IA, DB} PUSH (multiple registers) POP (multiple registers), LDC, STC, VLDM VSTM, VPUSH, VPOP, BLX, BLXNS, BX, BXNS VLLDM, VLSTM | Word |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XNVS}$ The following instructions change the endianness of data that is loaded or stored:

- REV
  Reverse word (four bytes) register, for transforming 32-bit representations.
- REVSH
  Reverse halfword and sign extend, for transforming signed 16-bit representations.
- REV16
  Reverse packed halfwords in a register for transforming unsigned 16-bit representations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B6.4 Alignment behavior

$R_{LKGV}$ All instruction fetches are halfword-aligned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RQGG}$ The following are unaligned data accesses that always generate an alignment fault:

- Non halfword-aligned LDAH, LDREXH, LDAEXH, STLH, STLEXH, and STREXH.
- Non word-aligned LDREX, LDAEX, STLEX, STREX, LDRD, LDMIA, LDMDB, POP (multiple registers), LDC, VLDR, VLDM, VPOP, LDA, STL, STMIA, STMDB, PUSH (mulitple registers), STC, VSTR, VSTM, VPUSG, VLLDM, and VLSTM.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MHCM}$ If CCR.UNALIGN_TRP is set to 1, the following are unaligned data accesses that generate an alignment fault:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JLGS}$ Unaligned accesses are only supported if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{WCVX}$ Accesses to Device memory are always aligned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PZTT}$ If the Main Extension is not implemented, unaligned accesses generate an alignment HardFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M**.*

$R_{RNDS}$ Alignment faults are synchronous and generate an UNALIGNED UsageFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BNBX}$ The CONSTRAINED UNPREDICTABLE behavior of unaligned loads and stores is one of the following:

- Generate an UNALIGNED UsageFault.
- Perform the specified load or store to the unaligned memory location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{LPVP}$ Unaligned loads and stores perform the specified load and store to the unaligned memory location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.14 *Normal memory* on page 208.

B6.16 *Device memory* on page 211.

# B6.5 Atomicity

## B6.5.1 Single-copy atomicity

$I_{NWVK}$ Store operations are *single-copy atomic* if, when they overlap bytes in memory:

1. All of the writes from one of the stores are inserted into the coherence order of each overlapping byte.
2. All of the writes from another of the stores are inserted into the coherence order of each overlapping byte.
3. Step 2 repeats, for each single-copy store atomic operation that overlaps.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BSHJ}$ The following data accesses are single-copy atomic:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QNPX}$ Instruction fetches are single-copy atomic at halfword granularity.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MXWC}$ For instructions that access a sequence of word-aligned words, each word access is single-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LKPM}$ For instructions that access a sequence of word-aligned words, the architecture does not require two or more subsequent word accesses to be single-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B6.5.2 Multi-copy atomicity

$I_{BCHK}$ In a multiprocessing environment, writes to memory are *multi-copy atomic* if all of the following are true:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GJGP}$ Writes to Normal memory are not required to be multi-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LBGB}$ Writes to Device memory with the Gathering attribute are not required to be multi-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WHJR}$ Writes to Device memory with the non-Gathering attribute that is single-copy atomic are also multi-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.16 *Device memory* on page 211.

B6.14 *Normal memory* on page 208.

B6.23 *Load-Acquire and Store-Release accesses to memory* on page 223.

## B6.6 Concurrent modification and execution of instructions

$I_{TFGC}$ The Armv8 architecture limits the set of instructions that can be executed by one thread of execution as they are being modified by another thread of execution without requiring explicit synchronization.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XWVK}$ Unless otherwise stated, concurrent modification and execution of instructions results in a CONSTRAINED UNPREDICTABLE choice of any behavior that can be achieved by executing any sequence of instructions from the same Security state or the same Privilege level.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BFPB}$ For instructions that can be concurrently modified, the PE executes either:

- The original instruction.
- The modified instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NNQK}$ A 16-bit instruction can be concurrently modified, where the 16-bit instruction before modification and the 16-bit modification is any of the following:

- B.
- BX.
- BLX.
- BKPT.
- NOP.
- SVC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KMZG}$ The hw1 of a 32-bit BL immediate instruction can be concurrently modified to the most significant halfword of another BL immediate instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HKGP}$ The hw1 of a 32-bit BL immediate instruction can be concurrently modified to a 16-bit B, BLX, BKPT, or SVC instruction. This modification also works in reverse.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FGBT}$ The hw2 of a 32-bit BL immediate instruction can be concurrently modified to the hw2 of another BL instruction with a different immediate.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NTVD}$ The hw2, of a 32-bit B immediate instruction with a condition field can be concurrently modified to the hw2 of another 32-bit B immediate instruction with a condition field with a different immediate.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CMZX}$ The hw2 of a 32-bit B immediate instruction without a condition field can be concurrently modified to the hw2 of another 32-bit B immediate instruction without a condition field.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.3 *Endianness* on page 188.

B.

`BL`.

`BLX, BLXNS`.

# B6.7 Access rights

$I_{JHGH}$ An instruction fetch or memory access is subject to the following checks in the following order:

1. Alignment.
2. SAU or IDAU or both.
3. MPU.
4. BusFault (IBUSERR).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TQJS}$ An exception is generated, instead of normal execution of the fetching and decoding process, if one of the following occurs.

| Priority | Fault type | Cause |
|---|---|---|
| Highest | One of the following Secure faults:<br>• INVEP<br>• INVTRAN | AU violation |
| ↓ | The following MemManage fault:<br>• IACCVIOL | MPU violation |
| ↓ | The following BusFault:<br>• IBUSERR | System fault |
| ↓ | One of the following:<br>• DebugMonitor exception<br>• Halted Debug Entry | FPB hit |
| ↓ | The following SecureFault:<br>• INVEP | SG check |
| ↓ | The following UsageFault:<br>• INVSTATE | T32 state check |
| Lowest | One of the following UsageFaults:<br>• UNDEFINSTR<br>• NOCP | Undefined instruction |

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a Secure fault requires S, a MemManage fault requires M && MPU, a Halted Debug Entry fault can only occur if Halting Debug is implemented, a DebugMonitor exception require DebugMonitor, UsageFault and BusFault require M, HardFault when !M.*

$R_{KPNQ}$ If a memory access fails its alignment check, the fetch is not presented to the SAU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{SDMQ}$ If an instruction fetch or memory access fails its AU check, the fetch is not presented to the relevant MPU for comparison.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S** && **MPU**.*

$R_{FLLN}$ If an instruction fetch or memory access fails its MPU check, it is not issued to the memory system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

## B6.8 Observability of memory accesses

$R_{PNDH}$ For a PE, the following mechanisms are treated as independent observers:

- The mechanism that performs reads from or writes to memory.
- The mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory. These accesses are treated as reads.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DVFW}$ The set of observers that can observe a memory access is not defined by the PE architecture.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{VSCK}$ In the context of observability, *subsequent* means whichever of the following descriptions is appropriate:

- After the point in time where the location is observed by the observer.
- After the point in time where the location is globally observed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VCCS}$ A write to a location in memory is *observed* by an observer when:

- A subsequent read of the location by the same observer would return the value that was written by the observed write or written by a write to that location by any observer that is sequenced in the coherence order of the location after the observed write.
- A subsequent write of the location by the same observer would be sequenced in the coherence order of the location after the observed write.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XQPT}$ A write to a location in memory is *globally observed* for a Shareability domain or set of observers when:

- A subsequent read of the location by any observer in that Shareability domain that is capable of observing the write would return the value that is written by the globally observed write or by a write to that location by any observer that is sequenced in the coherence order of the location after the globally observed write.
- A subsequent write to the location by any observer in that Shareability domain would be sequenced in the coherence order of the location after the globally observed write.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RSPX}$ For Device-nGnRnE memory, a read or write of a memory-mapped location in a peripheral is observed, and globally observed, only when the read or write:

- Meets the general observability conditions.
- Can begin to affect the state of the memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DGRR}$ A read of a location in memory is *observed* by an observer when a subsequent write to the location by the same observer would have no effect on the value that is returned by the read.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BVJF}$ A read of a location in memory is *globally observed* for a Shareability domain when a subsequent write to the location by any observer in that Shareability domain that is capable of observing the write would have no effect on the value that is returned by the read.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QRKX}$ Multiple writes to the same register will be observed in the same order by all observers. The architecture does not guarantee that all observers will observe all of the writes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HMHZ}$ Explicit synchronization is not required on an external read or write by an external agent to be observable to a following external read or write by that agent to the same register using the same address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TXSK}$ Explicit synchronization is not required for serial external accesses, either reads or writes, by a single external agent for any registers that are accessible as external system control registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.16 *Device memory* on page 211.

B6.17 *Device memory attributes* on page 213.

## B6.9 Completion of memory accesses

$R_{XCTL}$ A read or write is complete for a Shareability domain when the following conditions are true:

- The read or write is globally observed for that Shareability domain.
- All instruction fetches by observers within the Shareability domain have observed the read or write.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WCMQ}$ A cache or branch predictor maintenance instruction is complete for a Shareability domain when the effects of the instruction are globally observed for that Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SFLM}$ The completion of a memory access to Device memory other than Device-nGnRnE does not guarantee the visibility of the side-effects of the access to all observers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MWBK}$ The mechanism that ensures the visibility of the side-effects of the access to all observers is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.18 *Shareability domains* on page 216.

B6.16 *Device memory* on page 211.

B6.17 *Device memory attributes* on page 213.

## B6.10 Ordering requirements for memory accesses

$R_{RBDL}$ Armv8-M defines access restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses involved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GJDH}$ For all accesses to all memory types, the only stores by an observer that can be observed by another observer are those stores that have been architecturally executed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RXPL}$ Reads and writes can be observed in any order provided that, if an address dependency exists between two reads or between a read and a write, then those memory accesses are observed in program order by all observers within the common Shareability domain of the memory addresses being accessed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KWFG}$ Speculative writes by an observer cannot be observed by another observer.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VMHG}$ For Device memory with the non-Reordering attribute, memory accesses arrive at a single peripheral in program order.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WGCF}$ Memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by a context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RJMK}$ A register data dependency between the value that is returned by a load instruction and the address that is used by a subsequent memory transaction enforces an order between that load instruction and the subsequent memory transaction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.11 *Ordering of implicit memory accesses* on page 201.

B6.12 *Ordering of explicit memory accesses* on page 202.

B6.14 *Normal memory* on page 208.

B6.16 *Device memory* on page 211.

B6.18 *Shareability domains* on page 216.

## B6.11 Ordering of implicit memory accesses

$R_{KPFC}$ There are no ordering requirements for implicit accesses to any type of memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

## B6.12 Ordering of explicit memory accesses

$R_{BMNM}$ For all memory types, for accesses from a single observer, the requirements of uniprocessor semantics are maintained.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WTRP}$ For all types of memory, if there is a control dependency between a direct read and a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XGNP}$ For all types of memory, if the value returned by a direct read computes data that is written by a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MBNW}$ It is impossible for an observer to observe a write from a store that both:

- Has not been executed.
- Will not be executed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

B6.14 *Normal memory* on page 208.

B6.16 *Device memory* on page 211.

B6.17 *Device memory attributes* on page 213.

B6.18 *Shareability domains* on page 216.

B6.19 *Shareability attributes* on page 218.

# B6.13 Memory barriers

$R_{WRCT}$ The Arm architecture supports out-of-order completion of instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NBQC}$ Armv8 supports the following memory barriers:

- *Instruction Synchronization Barrier* (ISB).
- *Data Memory Barrier* (DMB).
- *Data Synchronization Barrier* (DSB).
- *Consumption of Speculative Data Barrier* (CSDB).
- *Physical Speculative Store Bypass Barrier* (PSSBB).
- *Speculative Store Bypass Barrier* (SSBB).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LQXF}$ The DMB and DSB memory barriers affect reads and writes to the memory system that are generated by Load/Store instructions and data or unified cache maintenance instructions that are executed by the PE. Instruction fetches are not explicit accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B6.13.1 Instruction Synchronization Barrier

$R_{STMG}$ An ISB ensures that all instructions that come after the ISB instruction in program order are fetched from the cache or memory after the ISB instruction has completed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

`InstructionSynchronizationBarrier()`.

Context synchronization event

## B6.13.2 Data Memory Barrier

$R_{MPSG}$ The required Shareability for a DMB is *Full system,* and applies to all observers in the Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GVDL}$ A DMB only affects memory accesses and the operation of data cache and unified cache maintenance instructions, and has no effect on the ordering of any other instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HFTX}$ A DMB that ensures the completion of cache maintenance instructions has an access type of both loads and stores.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WMRT}$ A DMB instruction creates two groups of memory accesses, Group A and Group B, and does not affect memory accesses that are in not in Group A or Group B:

**Group A** contains:

- All explicit memory accesses of the required access types from observers in the same Shareability domain as

PEe that are observed by PEe before the DMB instruction.
- All loads of required access types from an observer PEx in the same required Shareability domain as PEe that have been observed by any given different observer, PEy, in the same required Shareability domain as PEe before PEy 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 PEe that occur in program order after the DMB instruction.
- All explicit memory accesses of the required access types by any given observer PEx in the same required Shareability domain as PEe that can only occur after a load by PEx has returned the result of a store that is a member of Group B.

Any observer with the same required Shareability domain as PEe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the Shareability and Cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that are to be ordered are from the same PE, a DMB provides for this guarantee.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

`DataMemoryBarrier()`.

B6.18 *Shareability domains* on page 216.

### B6.13.3 Data Synchronization Barrier

$I_{CNFG}$ The DSB is a memory barrier that synchronizes the execution stream with memory accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NKWJ}$ The required Shareability for a DSB is Full system and applies to all observers in the Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VLBF}$ A DSB instruction creates two groups of memory accesses, Group A and Group B, and does not affect memory accesses that are in not in Group A or Group B:

**Group A** contains:

- All explicit memory accesses of the required access types from observers in the same Shareability domain as PEe that are observed by PEe before the DSB instruction.
- All loads of required access types from an observer PEx in the same required Shareability domain as PEe that have been observed by any given different observer, PEy, in the same required Shareability domain as PEe before PEy 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 PEe that occur in program order after the DSB instruction.
- All explicit memory accesses of the required access types by any given observer PEx in the same required Shareability domain as PEe that can only occur after a load by PEx has returned the result of a store that is a member of Group B.

Any observer with the same required Shareability domain as PEe observes all members of Group A before it observes any member of Group B to the extent that those group members are required to be observed, as determined by the Shareability and Cacheability of the memory addresses accessed by the group members.

If members of Group A and members of Group B access the same memory-mapped peripheral of arbitrary system-defined size, then members of Group A that are accessing Device or Normal Non-cacheable memory arrive at that peripheral before members of Group B that are accessing Device or Normal Non-cacheable memory. Where the members of Group A and Group B that are to be ordered are from the same PE, a DSB provides for this guarantee.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KMGH}$ A DSB completes when all of the following conditions apply:

- All explicit memory accesses that are observed by PEe before the DSB is executed and are of the required access types, and are from observers in the same required Shareability domain as PEe, are complete for the set of observers in the required Shareability domain.
- If the required access types of the DSB is reads and writes, then all cache and branch predictor maintenance instructions that are issued by PEe before the DSB are complete for the required Shareability domain.
- All explicit accesses to the System Control Space that result in a context altering operation issued by PEe before the DSB are complete.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KMBX}$ No instruction that appears in program order after the DSB instruction can execute until the DSB completes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

`DataSynchronizationBarrier()`.

B6.18 *Shareability domains* on page 216.

### B6.13.4 Consumption of Speculative Data Barrier

$R_{CTSR}$ The CSDB is a memory barrier that prevents instructions that appear in program order after the barrier completes from determining any part of the value of data derived from speculatively-executed load instructions that appeared in program order before completion of the CSDB memory barrier.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{LZDK}$ When a CSDB instruction is executed but before the CSDB completes and there are three instructions:

1. A load instruction speculatively executed in program order before the barrier that might or might not be architecturally executed.
2. A Conditional Move instruction that has passed its condition code check and does not have an address dependency for an input register on the speculatively-executed load.
3. A load, store, data or instruction preload appearing in program order after the barrier, which has an address dependency on the Conditional Move instruction.

The speculative execution of the load, store, data or instruction preload does not influence the allocation of cache entries to determine any part of the value of the speculatively executed load instruction by an evaluation of the cache entries which have been allocated or evicted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{DDTH}$ When a CSDB instruction is executed but before the CSDB completes and there are three instructions:

1. A load instruction speculatively executed in program order before the barrier that might or might not be architecturally executed.
2. A Conditional Move instruction that has no dependency to pass the condition tests or for an input register on the speculatively executed load.
3. An indirect branch instruction, appearing in program order after the barrier, that is dependent on the Conditional Move instruction for the target address of the indirect branch.

The speculative execution of the indirect branch does not influence the allocation of cache entries to determine any part of the value of the speculatively executed load instruction by an evaluation of the cache entries which have been allocated or evicted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{JWCV}$ A CSDB instruction cannot be executed speculatively.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{QZKB}$ A CSDB can be inserted speculatively and completed when it is known not to be speculative.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{WGCX}$ The CSDB instruction is not available in an implementation without the Main Extension.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, !M.*

$I_{PCSF}$ Arm recommends that a combination of DSB SYS and an ISB is inserted to prevent consumption of speculative data.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, !M.*

### B6.13.5 Physical Speculative Store Bypass Barrier

$I_{CCNK}$ The PSSBB prevents speculative loads from:

- Returning data older than the most recent store to the same physical address appearing in program order before the load.
- Returning data from stores using the same physical address appearing in program order after the load.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{MDLZ}$ The PSSBB is not available in an implementation without the Main Extension.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, !M.*

### B6.13.6 Speculative Store Bypass Barrier

$I_{HWND}$ The SSBB prevents speculative loads from:

- Returning data older than the most recent store to the same address appearing in program order before the load.
- Returning data from stores using the same address appearing in program order after the load.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{MGLH}$ The SSBB is not available in an implementation without the Main Extension.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, !M.*

### B6.13.7 Synchronization requirements for System Control Space

$R_{SJQJ}$ A DSB guarantees that all writes to the System Control Space have been completed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NPDJ}$ A DSB does not guarantee that the side effects of writes to the System Control Space are visible.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HMNM}$ A Context synchronization event guarantees that the side effects of any completed writes to the System Control Space are visible after the Context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B7.3 *The System Control Space (SCS)* on page 245.

## B6.14 Normal memory

$I_{NVRF}$ Memory locations that are *idempotent* have the following properties:

- Read accesses can be repeated with no side-effects.
- Repeated read accesses return the last value that is written to the resource being read.
- Read accesses can fetch additional memory locations with no side-effects.
- Write accesses can be repeated with no side-effects, if the contents of the location that is accessed are unchanged between the repeated writes or as the result of an exception.
- Unaligned accesses can be supported.
- Accesses can be merged before accessing the target memory system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QGCF}$ The PE is permitted to treat regions of memory assigned the memory type Normal memory as idempotent.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CGJX}$ Normal memory can be marked as Cacheable or Non-cacheable. Normal memory is assigned Cacheability attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LCPJ}$ Normal Non-cacheable memory is always treated as shareable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PKXL}$ Speculative data accesses to Normal memory are permitted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WLVR}$ A write to Normal memory completes in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WLCV}$ A write to a Non-cacheable Normal memory location reaches the endpoint for that location in the memory system in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MJWF}$ A completed write to Normal memory is globally observed for the *Shareability domain* in finite time without the requirement for cache maintenance instructions or memory barriers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NHFQ}$ For multi-register Load/Store instructions that access Normal memory, the architecture does not define the order in which the registers are accessed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CFHV}$ There is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

B6.18 *Shareability domains* on page 216.

B6.15 *Cacheability attributes* on page 210.

B6.22 *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page 222.

*MAIR_ATTR, Memory Attributes Indirection Register Attributes*.

## B6.15 Cacheability attributes

$R_{KXJV}$ The architecture provides Cacheability attributes that are defined independently for each of two conceptual levels of cache:

- The Inner cache.
- The Outer cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XRWS}$ The Cacheability attributes are:

- Non-cacheable.
- Write-Through Cacheable.
- Write-Back Cacheable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XQXW}$ It is IMPLEMENTATION DEFINED whether Write-Through Cacheable and Write-Back Cacheable can have the additional attribute Transient or Non-transient.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{LDXP}$ The Transient attribute is a memory hint that indicates that the benefit of caching is for a short period. The architecture does not define what is meant by a *short period*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CFKN}$ Cacheability attributes other than Non-cacheable can be complemented by the following cache allocation hints, which are independent for read and write accesses:

- Read-Allocate, Transient Read-Allocate, or No Read-Allocate.
- Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DRTR}$ The architecture does not require an implementation to make any use of cache allocation hints.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FQSS}$ Any cacheable Normal memory region is treated as Read-Allocate, No Write-Allocate unless it is explicitly assigned other cache allocation hints.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{FRVF}$ A Cacheable location with no Read-Allocate and no Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply to a location that is Cacheable, no Read-Allocate, no Write-Allocate.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FTKW}$ All data accesses to Non-cacheable Normal memory locations are data coherent to all observers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.14 *Normal memory* on page 208.

## B6.16 Device memory

$I_{BXHS}$ Device memory is a *memory type* that is assigned to regions of memory where accesses can have side-effects.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WTZL}$ Device memory is not cacheable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LDDN}$ Device memory is always treated as shareable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PQXS}$ Speculative data accesses cannot be made to Device memory. However, for instructions that access a sequence of word-aligned words, the accesses might occur multiple times.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NLHC}$ Speculative instruction fetches can be made to Device memory, unless the location is marked as execute-never.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CSKG}$ Any unaligned access to Device memory generates an UNALIGNED UsageFault exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{YMTK}$ Device memory is assigned a combination of *Device memory attributes*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LFTG}$ A write to Device memory completes in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FSCD}$ A write to a Device memory location reaches the endpoint for that location in the memory system in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GTTQ}$ A completed write to a Device memory location is globally observed for the Shareability domain in finite time without the requirement for cache maintenance instructions or barriers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XMCH}$ If the content of a Device memory location changes without a direct write to the location, the change is observed for the Shareability domain in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KJHG}$ For an instruction fetch from Device memory, if a branch causes the Program Counter to point to an area of memory that is not marked as Execute-never, the implementation can either:

- Treat the fetch as if it is to a location in Normal Non-cacheable memory.
- Take an IACCVIOL MemManage fault.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a MemManage fault requires M.*

$R_{DFJX}$ There is no requirement for the memory system beyond the PE to be able to identify the size of the elements that are accessed, for instructions that load the following from Device memory:

- More than one general-purpose register.
- One or more registers from the floating-point register file.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**$R_{KVHT}$** For an `LDM`, `STM`, `LDRD`, or `STRD` instruction with a register list that includes the PC, the architecture does not define the order in which the registers are accessed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

**$R_{SFPK}$** For an `LDM`, `STM`, `VLDM`, or `VSTM` instruction with a register list that does not include the PC, all registers are accessed in the order that they appear in the register list, for Device memory with the non-Reordering attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

B6.19 *Shareability attributes* on page 218.

B6.17 *Device memory attributes* on page 213.

B6.18 *Shareability domains* on page 216.

# B6.17 Device memory attributes

$R_{VNSJ}$ Each Device memory region is assigned a combination of Device memory attributes. The attributes are:

**Gathering, G and nG**: The *Gathering* and *non-Gathering* attributes.

**Reordering, R and nR**: The *Reordering* and *non-Reordering* attributes.

**Early Write Acknowledgement, E and nE**: The *Early Write Acknowledgement* and *no Early Write Acknowledgement* attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CFFC}$ Each Device memory region is assigned one of the combinations in the following table:

| Memory Ordering | Name | nG | nR | nE | G | R | E |
|---|---|---|---|---|---|---|---|
| Strong | Device-nGnRnE | Y | Y | Y | - | - | - |
| ↓ | Device-nGnRE | Y | Y | - | - | - | Y |
| ↓ | Device-nGRE | Y | - | - | - | Y | Y |
| Weak | Device-GRE | - | - | - | Y | Y | Y |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LJKD}$ Weaker memory can be accessed according to the rules specified for stronger memory:

- Memory with the:
  - G attribute can be accessed according to the rules specified for the nG attribute.
  - nG attribute cannot be accessed according to the rules specified for the G attribute.
- Memory with the:
  - R attribute can be accessed according to the rules specified for the nR attribute.
  - nR attribute cannot be accessed according to the rules specified for the R attribute.

Because the nE attribute is a hint:

- An implementation is permitted to perform an access with the E attribute in a manner consistent with the requirements specified by the nE attribute.
- An implementation is permitted to perform an access with the nE attribute in a manner consistent with the relaxations allowed by the E attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FJXX}$ For Device-GRE and Device-nGRE memory, the use of barriers is required to order accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PVCY}$ Memory accesses that are generated by vector instructions that target any type of Device memory operate as if the access had targeted a Device-GRE region.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B6.17.1 *Gathering and non-Gathering Device memory attributes* on page 214.

B6.17.2 *Reordering and non-Reordering Device memory attributes* on page 214.

B6.17.3 *Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes* on page 215.

B6.16 *Device memory* on page 211.

### B6.17.1 Gathering and non-Gathering Device memory attributes

#### G attribute

$R_{DBSX}$ If multiple accesses of the same type, read or write, are to:

- The same location, with the G attribute, they can be merged into a single transaction.
- Different locations, all with the G attribute, they can be merged into a single transaction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KCMX}$ Gathering of accesses that are separated by a memory barrier is not permitted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JSRD}$ Gathering of accesses that are generated by a Load-Acquire/Store-Release is not permitted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MGKJ}$ A read can come from intermediate buffering of a previous write if:

- The accesses are not separated by a DMB or DSB barrier.
- The accesses are not separated by any other ordering construction that requires that the accesses are in order, for example a combination of Load-Acquire and Store-Release.
- The accesses are not generated by a Store-Release instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{SRDS}$ The architecture only defines programmer visible behavior. Therefore, if a programmer cannot tell whether Gathering has occurred, Gathering can be performed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

#### nG attribute

$R_{GVTF}$ Multiple accesses to a memory location with the nG attribute cannot be merged into a single transaction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BTWD}$ A read of a memory location with the nG attribute cannot come from a cache or a buffer, but comes from the endpoint for that address in the memory system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.23 *Load-Acquire and Store-Release accesses to memory* on page 223.

### B6.17.2 Reordering and non-Reordering Device memory attributes

#### R attribute

$R_{RPTB}$ This attribute imposes no restrictions or relaxations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

#### nR attribute

$R_{DFXL}$ If the access is to a:

- Peripheral, it arrives at the peripheral in program order. If there is a mixture of accesses to Device nGnRE and Device-nGnRnE in the same peripheral, these accesses occur in program order.
- Non-peripheral, this attribute imposes no restrictions or relaxations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BDWB}$ The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee that is provided by the DMB instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NDHC}$ The non-Reordering attribute does not require any additional ordering, other than the ordering that applies to Normal memory, between:

- Accesses with the non-Reordering attribute and accesses with the Reordering attribute.
- Accesses with the non-Reordering attribute and accesses to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### B6.17.3 Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes

#### E attribute

$R_{PVSH}$ The E attribute imposes no restrictions or relaxations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

#### nE attribute

$R_{FWFR}$ Assigning the nE attribute recommends that only the endpoint of the write access returns a write acknowledgement of the access, and that no earlier point in the memory system returns a write acknowledgement.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{FQWQ}$ The E attribute is treated as a hint. Arm strongly recommends that this hint is not ignored by a PE, but is made available for use by the system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.13 *Memory barriers* on page 203.

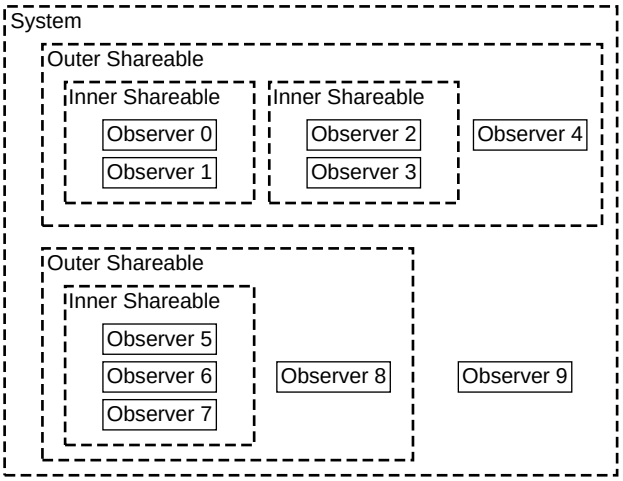
## B6.18 Shareability domains

$R_{JMHL}$ There are two conceptual Shareability domains:

- The Inner Shareability domain.
- The Outer Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{XQWM}$ The following diagram shows the Shareability domains:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MCPS}$ All observers in an Inner Shareability domain are data coherent for data accesses to memory that has the *Inner-shareable Shareability attribute*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SVCR}$ All observers in an Outer Shareability domain are data coherent for data accesses to memory that has the *Outer-shareable Shareability attribute*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JMFS}$ Each observer is a member of only a single Inner Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BNWH}$ Each observer is a member of only a single Outer Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FVBG}$ All members of the same Inner Shareability domain are always members of the same Outer Shareability domain.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WFMV}$ Accesses to a shareable memory location are coherent within the Shareability domain of that location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DHJF}$ An Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper

subset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XHJL}$ Hardware is required to ensure coherency and ordering within the Shareability domain if all of the following apply:

- Before writing to a location not using the Write-Back attribute, a location in the caches that might have been written with the Write-Back attribute by an agent has been invalidated or cleaned.
- After writing the location with the Write-Back attribute, the location has been cleaned from the caches to make the write visible to external memory.
- Before reading the location with a cacheable attribute, the cache location has been invalidated, or cleaned and invalidated.
- A `DMB` barrier instruction has been executed, with a scope that applies to the common Shareability of the accesses, between any accesses to the same memory location that use different attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.8 *Observability of memory accesses* on page 197.

B6.19 *Shareability attributes* on page 218.

## B6.19 Shareability attributes

$R_{CJRF}$ Each Normal cacheable memory region is assigned one of the following Shareability attributes:

- *Non-shareable*.
- *Inner-shareable*.
- *Outer-shareable*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PDVV}$ For Non-shareable memory, hardware is not required to make data accesses by different observers coherent. If a number of observers share the memory, cache maintenance instructions, in addition to the barrier operations that are required to ensure memory ordering, can ensure that the presence of caches does not lead to coherency issues.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XTVD}$ Non-cacheable Normal memory locations are always treated as Outer Shareable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

B6.14 *Normal memory* on page 208.

B6.16 *Device memory* on page 211.

B6.18 *Shareability domains* on page 216.

B6.32 *Cache maintenance operations* on page 234.

## B6.20 Memory access restrictions

$R_{KSXT}$ For accesses to any two bytes that are accessed by the same instruction, the two bytes have the same memory type and Shareability attributes, otherwise behavior is a CONSTRAINED UNPREDICTABLE choice of the following:

- All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the first address that is accessed by the instruction.
- All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the last address that is accessed by the instruction.
- Each memory access that is generated by the instruction uses the memory type and Shareability attribute that is associated with its own address.
- The instruction executes as a NOP.
- The instruction generates an alignment fault caused by the memory type.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{WRBT}$ Except for possible differences in cache allocation hints, Arm deprecates having different Cacheability attributes for accesses to any two bytes that are generated by the same instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BFKS}$ If the accesses of an instruction that cause multiple accesses to any type of Device memory cross the boundary of a memory region then the behavior is a CONSTRAINED UNPREDICTABLE choice of the following:

- All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses.
- All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses, except that there is no guarantee of ordering between memory accesses,
- The instruction executes as a NOP.
- The instruction generates an alignment fault caused by the memory type.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.1 *Memory accesses* on page 186.

## B6.21 Mismatched memory attributes

$R_{XHTK}$ Memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all the following memory attributes of that location:

- Memory type - Device or Normal.
- Shareability.
- Cacheability, for the same level of the Inner or Outer cache, but excluding any cache allocation hints.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VKHJ}$ When a memory location is accessed with mismatched attributes, the only permitted effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
  - A read of the memory location by one agent might not return the value that was most recently written to that memory location by the same agent.
  - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory location.
- There might be a loss of the properties that are derived from the memory type.
- If all Load-Exclusive/Store-Exclusive instructions that are executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes that are written without the Write-Back cacheable attribute and that are within the same Write-Back granule as bytes that are written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NJLB}$ The loss of the properties that are associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Reordering.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QCKK}$ If the only memory type mismatch that is associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HCCD}$ Any agent that reads a memory location with mismatched attributes using the same common definition of the Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:

- All aliases to the memory location with write permission both use a common definition of the Shareability and Cacheability attributes for the memory location, and have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
- All aliases to a memory location use a definition of the Shareability attributes that encompasses all the agents with permission to access the location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GBKH}$ The possible permitted effects that are caused by mismatched attributes for a memory location are defined more precisely if all the mismatched attributes define the memory location as one of:

- Any Device memory type.
- Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties that are derived from the memory type when multiple agents attempt to access the memory location.
- Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting DMB barrier instructions between accesses to the same memory location that might use different attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VVBS}$ If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:

- Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
- A DMB barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory location that use different attributes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VCXW}$ If multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{TPWG}$ Arm strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

B6.18 *Shareability domains* on page 216.

B6.15 *Cacheability attributes* on page 210.

B6.16 *Device memory* on page 211.

B6.14 *Normal memory* on page 208.

B6.22 *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page 222.

## B6.22 Load-Exclusive and Store-Exclusive accesses to Normal memory

$R_{KDWC}$ For Normal memory that is:

- Non-shareable, it is IMPLEMENTATION DEFINED whether Load-Exclusive and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.
- Shareable, Load-Exclusive, and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.14 *Normal memory* on page 208.

B6.1 *Memory accesses* on page 186.

## B6.23 Load-Acquire and Store-Release accesses to memory

$I_{VVTX}$ The following table summarizes the Load-Acquire/Store-Release instructions.

| Data type | Load- Acquire | Store- Release | Load-Acquire Exclusive | Store-Release Exclusive |
|---|---|---|---|---|
| 32-bit word | LDA | STL | LDAEX | STLEX |
| 16-bit halfword | LDAH | STLH | LDAEXH | STLEXH |
| 8-bit byte | LDAB | STLB | LDAEXB | STLEXB |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XBRM}$ A Store-Release followed by a Load-Acquire is observed in program order by each observer within the Shareability domain of the memory address being accessed by the Store-Release and the memory address being accessed by the Load-Acquire.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RRFK}$ For a Load-Acquire, observers in the Shareability domain of the address that is accessed by the Load-Acquire observe accesses in the following order:

1. The read caused by the Load-Acquire.
2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the Shareability of the address that is accessed by the load or store requires that the observer observes the access.

There are no other ordering requirements on loads or stores that appear before the Load-Acquire.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WLWT}$ For a Store-Release, observers in the Shareability domain of the address that is accessed by the Store-Release observe accesses in the following order:

1. All of the following for which the Shareability of the address that is accessed requires that the observer observes the access:
   - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
   - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
2. The write caused by the Store-Release.

There are no other ordering requirements on loads or stores that appear in program order after the Store-Release.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HCKC}$ All Store-Release instructions are multi-copy atomic when they are observed with Load-Acquire instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DGXR}$ A Load-Acquire to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CKRC}$ A Store-Release to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined

as any type of Device memory access ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GJHK}$ If a Load-Acquire to a memory address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access has observed the value that is stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WRLC}$ Load-Acquire and Store-Release access only a single data element.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KCTN}$ Load-Acquire and Store-Release accesses are single-copy atomic.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BXRP}$ If a Load-Acquire or Store-Release instruction accesses an address that is not aligned to the size of the data element being accessed, the access generates an alignment fault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NVRJ}$ A Store-Release Exclusive instruction only has the release semantics if the store is successful.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.18 *Shareability domains* on page 216.

B6.16 *Device memory* on page 211.

## B6.24 Caches

$I_{JSPB}$ When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache can depend on many aspects of the implementation, such as the following factors:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QGSG}$ An implementation can include multiple levels of cache, up to a maximum of seven levels, in a hierarchical memory system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{STRV}$ The lower the cache level, the closer the cache is to the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PDSR}$ Entries for addresses with a Normal cacheable attribute can be allocated to an enabled cache at any time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JGBL}$ The allocation of a memory address to a cache location is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SBGJ}$ A cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XXBW}$ Where a breakdown in coherency can occur, data coherency of the caches is controlled in an IMPLEMENTATION DEFINED manner.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JVJN}$ The architecture cannot guarantee whether:

- A memory location that is present in the cache remains in the cache.
- A memory location that is not present in the cache is brought into the cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PHWM}$ If the cache is disabled, no new allocation of memory locations into the cache occurs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LJQB}$ The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it had previously been visible to that observer.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QRLS}$ If the cache is enabled, it is guaranteed that no memory location that does not have a cacheable attribute is allocated into the cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XXVH}$ If the cache is enabled, it is guaranteed that no memory location is allocated to the cache if the access permissions for that location are so that the location cannot be accessed by reads and cannot be accessed by writes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SCKQ}$ Any cached memory location is not guaranteed to remain incoherent with the rest of memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RQXN}$ If an implementation permits cache hits when the Cacheability control fields force all memory locations to be treated as Non-cacheable, then the cache initialization routine:

- Provides a mechanism to ensure the correct initialization of the caches.
- Is documented clearly as part of the documentation of the device.

In particular, if an implementation permits cache hits when the Cacheability controls force all memory locations to be treated as Non-cacheable, and the cache contents are not invalidated at reset, the initialization routine avoids any possibility of running from an uninitialized cache. It is acceptable for an initialization routine to require a fixed instruction sequence to be placed in a restricted range of memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WDBP}$ It is UNPREDICTABLE whether the location is returned from cache or from memory when:

- The location is not marked as cacheable but is contained in the cache. This situation can occur if a location is marked as Non-cacheable after it has been allocated into the cache.
- The location is marked as cacheable and might be contained in the cache, but the cache is disabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NDNN}$ The architecture allows copies of control values or data values to be cached. The existence of such copies can lead to CONSTRAINED UNPREDICTABLE behavior, if the cache has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with:

- The old value.
- The new value.
- An amalgamation of the old and new values.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BMPQ}$ The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.25 *Cache identification* on page 227.

B6.28 *Cache enabling and disabling* on page 230.

B6.15 *Cacheability attributes* on page 210.

B6.29 *Cache behavior at reset* on page 231.

B6.33 *Ordering of cache maintenance operations* on page 238.

B6.21 *Mismatched memory attributes* on page 220.

## B6.25 Cache identification

$R_{WBGH}$ A PE controls the implemented caches using:

- A single Cache Type Register, CTR.
- A single Cache Level ID Register, CLIDR.
- A single Cache Size Selection Register, CSSELR.
- For each implemented cache, across all levels of caching, a Cache Size Identification Register, CCSIDR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XJTL}$ The number of levels of cache is IMPLEMENTATION DEFINED and can be determined from the Cache Level ID Register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{PPSB}$ Cache sets and Cache ways are numbered from 0. Usually the set number is an IMPLEMENTATION DEFINED function of an address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## B6.26 Cache visibility

$R_{QLVB}$ A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RCHC}$ A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.15 *Cacheability attributes* on page 210.

## B6.27 Cache coherency

$R_{NNDJ}$ Data coherency of caches is ensured:

- When caches are not used.
- As a result of cache maintenance operations.
- By the use of hardware coherency mechanisms to ensure coherency of data accesses to memory for cacheable locations by observers in different Shareability domains.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CPGW}$ Hardware is not required to ensure coherency between instruction caches and memory, even for regions of memory with the Shareability attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.32 *Cache maintenance operations* on page 234.

B6.13 *Memory barriers* on page 203.

B6.19 *Shareability attributes* on page 218.

## B6.28 Cache enabling and disabling

$I_{PPLL}$ The Configuration and Control Register, CCR, enables and disables caches across all levels of cache that are visible to the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HTLD}$ It is IMPLEMENTATION DEFINED whether the CCR.DC and CCR.IC bits affect the memory attributes that are generated by an enabled MPU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **MPU**.*

$I_{TNHX}$ An implementation can use control bits in the Auxiliary Control Register, ACTLR, for finer-grained control of cache enabling.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SMDL}$ For instruction fetches and data accesses, NS-Attr determines which banked instance, either Secure or Non-secure, of CCR.IC or CCR.DC is used.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DSTQ}$ If the MPU is disabled, MPU_CTRL.ENABLE == 0, the CCR.DC and CCR.IC bits determine the cache state for cacheable regions of the default address map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **MPU**.*

See also:

B6.25 *Cache identification* on page 227.

B6.24 *Caches* on page 225.

B6.29 *Cache behavior at reset* on page 231.

B3.14 *Secure address protection* on page 95.

## B6.29 Cache behavior at reset

$R_{KCFK}$ All caches are disabled at reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JMBT}$ An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled:

- The exact form of any required cache initialization routine is IMPLEMENTATION DEFINED.
- If a required initialization routine is not performed, the state of an enabled cache is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TVKQ}$ If an implementation permits cache hits when the cache is disabled, the cache initialization routine provides a mechanism to ensure the correct initialization of the caches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CJGV}$ If an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine avoids any possibility of running from an uninitialized cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JSQQ}$ An initialization routine can require a fixed instruction sequence to be placed in a restricted range of memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JCTD}$ Arm recommends that whenever an invalidation routine is required, it is based on the Armv8-M cache maintenance operations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.24 *Caches* on page 225.

B6.28 *Cache enabling and disabling* on page 230.

B6.32 *Cache maintenance operations* on page 234.

# B6.30 Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches

$I_{CQLR}$ `PLD` and `PLI` are memory system hints and their effect is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{TPPK}$ The instructions `PLD` and `PLI` do not generate exceptions but the memory system operations might generate an imprecise fault (asynchronous exception) because of the memory access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QNGJ}$ A `PLD` instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the equivalent load from the same location with the same context and at the same privilege level and Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SFNK}$ A `PLD` instruction does not access Device-nGnRnE or Device-nGnRE memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HNLN}$ A `PLI` instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the fetch resulting from changing the PC to the location specified by the PLI instruction with the same context and at the same privilege level and Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MRFG}$ A `PLI` instruction cannot access memory that has the Device-nGnRnE or Device-nGnRE attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

`PLD, PLDW (immediate)`.

`PLD (literal)`.

`PLD, PLDW (register)`.

`PLI (immediate, literal)`.

`PLI (register)`.

# B6.31 Branch predictors

$I_{GTPB}$ Branch predictor hardware typically uses a form of cache to hold branch information.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MTBD}$ Branch predictors are not architecturally visible.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{CVCV}$ The BPIALL operation is provided for timing and determinism

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.34 *Branch predictor maintenance operations* on page 239.

## B6.32 Cache maintenance operations

$I_{MRMG}$ Cache maintenance operations act on particular memory locations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JJLL}$ Following a Clean operation, updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the operation is performed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VRBP}$ The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SJFS}$ Following an invalidate operation, updates made visible by observers that access memory at the point to which the invalidate is defined are made visible to an observer that controls the cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PGXK}$ An invalidate operation might result in the loss of updates to the locations affected by the operation that have been written by observers that access the cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TKBD}$ If the address of an entry on which the invalidate operates does not have a Normal cacheable attribute, or if the cache is disabled, then an invalidate operation ensures that this address is not present in the cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JTXK}$ If the address of an entry on which the invalidate operates has the Normal cacheable attribute, the cache invalidate operation cannot ensure that the address is not present in an enabled cache.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SDVP}$ A clean and invalidate operation behaves as the execution of a clean operation followed immediately by an invalidate operation. Both operations are performed to the same location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VKSN}$ The clean operation cleans from the level of cache that is specified through at least the next level of cache away from the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GFXB}$ The invalidate operation invalidates only at the level specified.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KVSM}$ For set/way operations and for All (entire cache) operations, the cache maintenance operation is to the next level of caching.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JTWT}$ For address operations, the cache maintenance operation is to the point of coherency (PoC) or to the point of unification (PoU) depending on the settings in CLIDR.{LoC,LOUU}.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XLHX}$ Data cache maintenance operations affect data caches and unified caches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QKMF}$ Instruction cache maintenance operations only affect instruction caches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RSVL}$ Cache maintenance operations are memory mapped, 32-bit write-only operations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NSHH}$ Cache maintenance operations can have one of the following side-effects:

- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

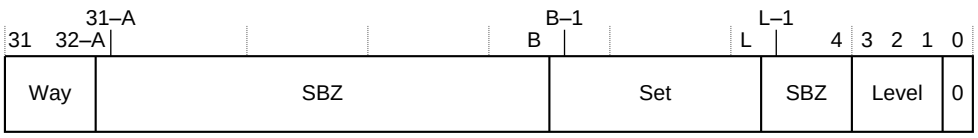
$R_{DWMR}$ The `ICIMVAU`, `DCIMVAC`, `DCCMVAU`, `DCCMVAC`, and `DCCIMVAC` operations require the physical address in the memory map but it does not have to be cache-line aligned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HCTC}$ For `DCISW`, `DCCSW`, and `DCCISW`, the `STR` operation identifies the cache line to which it applies by specifying the following:

- The cache set the line belongs to.
- The way number of the line in the set.
- The cache level.

The format of the register data for a set/way operation is:




Where:

**A** = Log2(ASSOCIATIVITY), rounded up to the next integer if necessary.

**B** = (L + S).

**L** = Log2(LINELEN).

**S** = Log2(NSETS), rounded up to the next integer if necessary. ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on.

The values of A and S are rounded up to the next integer.

**Level**= ((Cache level to operate on)-1). For example, this field is 0 for operations on an L1 cache, or 1 for operations on an L2 cache.

**Set** = The number of the set to operate on.

**Way** = The number of the way to operate on.

- If L == 4 then there is no SBZ field between the set and level fields in the register.
- If A == 0 there is no way field in the register, and register bits[31:B] are SBZ.

- If the level, set, or way field in the register is larger than the size implemented in the cache, then the effect of the operation is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RSBX}$ After the completion of an instruction cache maintenance operation, a context synchronization event guarantees that the effects of the cache maintenance operation are visible to all instruction fetches that follow the context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DHJQ}$ Arm recommends that, wherever possible, all caches that require maintenance to ensure coherency are included in the caches affected by the architecturally-defined cache maintenance operations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LRGS}$ It is IMPLEMENTATION DEFINED whether the `DCIMVAC` and `DCISW` operations, when performed from Non-secure state either:

- Clean any data that might be Secure data before invalidating it.
- Do not invalidate Secure data.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{VKDF}$ `ICIALLU`, `ICIMVAU`, `DCCMVAU`, `DCCMVAC`, `DCCSW`, `DCCIMVAC`, `DCCISW`, and `BPIALL` operations on Secure data might be ignored if the operation was performed from Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{MLLC}$ The following is the sequence of cache cleaning operations for a line of self-modifying code.

```
; Enter this code with <Rx> containing the new 32-bit instruction and <Ry>;
containing the address of the instruction.
; Use STRH in the first line instead of STR for a 16-bit instruction.
STR <Rx>, [<Ry>] ; Write instruction to memory
DSB ; Ensure write is visible
MOV <Rt>, 0xE000E000 ; Create pointer to base of System Control Space
STR <Ry>, [<Rt>,#0xF64] ; Clean data cache by address to point of unification
DSB ; Ensure visibility of the data cleaned from the cache
STR <Ry>, [<Rt>,#0xF58] ; Invalidate instruction cache by address to PoU
STR <Ry>, [<Rt>,#0xF78] ; Invalidate branch predictor
DSB ; Ensure completion of the invalidations
ISB ; Synchronize fetched instruction stream
```

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HXMM}$ If the Security attribution of memory is changed, it is IMPLEMENTATION DEFINED whether cache maintenance operations are required to keep the system state valid.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{JFGF}$ In the cache maintenance instructions that operate by Set/Way, if any index argument is larger than the value supported by the implementation, then the behavior is CONSTRAINED UNPREDICTABLE and one of the following occurs:

- The instruction generates a BusFault.
- The instruction performs cache maintenance on one of the following:
  - No cache lines.
  - A single arbitrary cache line.

– Multiple arbitrary cache lines.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a BusFault requires M.*

See also:

Cache Maintenance Operations.

Cache Maintenance Operations (NS alias).

B6.8 *Observability of memory accesses* on page 197.

B6.15 *Cacheability attributes* on page 210.

## B6.33 Ordering of cache maintenance operations

$R_{GCNB}$ All cache and branch predictor maintenance operations that do not specify an address execute, relative to each other, in program order.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GXNL}$ All cache maintenance operations that specify an address:

- Execute in program order relative to all cache operations that do not specify an address.
- Execute in program order relative to all cache maintenance operations that specify the same address.
- Can execute in any order relative to cache maintenance operations that specify a different address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RTJG}$ There is no restriction on the ordering of data or unified cache maintenance operation by address relative to any explicit load or store.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MJPP}$ There is no restriction on the ordering of a data or unified cache maintenance operation by set/way relative to any explicit load or store.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{VXXZ}$ A `DSB` instruction can be inserted to enforce ordering as required.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SWBG}$ For the `ICIALLU` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{ZQQZ}$ In a PE with the Security Extension, if cache maintenance operations are required when the security attribution of memory is changed, the following sequence of steps can be followed:

1. If the attribution of the address range changes from Secure to Non-secure, ensure that memory does not contain any data that is to remain secure.
2. Execute a `DSB` instruction.
3. Clean the affected lines in data or unified caches using the `DCC*` instruction.
4. Execute a `DSB` instruction.
5. Change the security attribution of the address range.
6. Execute a `DSB` instruction.
7. Invalidate the affected lines in all caches using the `DCI*` and `ICI*` instructions.
8. Execute a Context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.13.3 *Data Synchronization Barrier* on page 204.

B9.2 *Security attribution* on page 261.

B6.32 *Cache maintenance operations* on page 234.

## B6.34 Branch predictor maintenance operations

$R_{HVXX}$ Branch predictor maintenance operations are independent of cache maintenance operations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NSRK}$ A Context synchronization event event that follows a branch predictor maintenance operation guarantees that the effects of the branch predictor maintenance operation are visible to all instructions after the Context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HRXF}$ For the `BPIALL` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LXHX}$ As a side-effect of a branch predictor maintenance operation, any entry in the branch predictor might be invalidated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

Cache Maintenance Operations.

Cache Maintenance Operations (NS alias).

*BPIALL, Branch Predictor Invalidate All*.

B6.13 *Memory barriers* on page 203.

`DSB`.

# Chapter B7
# The System Address Map

This chapter specifies the Armv8-M system address map rules. It contains the following sections:

# B7.1 System address map

$R_{FQSD}$ The address space is divided into the following regions:

| Address | Region | Memory type | XN? | Cache | Shareability | Example usage |
|---|---|---|---|---|---|---|
| 0x00000000 - 0x1FFFFFFF | Code | Normal | - | WT RA | Non-shareable | Typically ROM or flash usage |
| 0x20000000- 0x3FFFFFFF | SRAM | Normal | - | WBWA RA | Non-shareable | SRAM region typically used for on-chip RAM. |
| 0x40000000- 0x5FFFFFFF | Peripheral | Device, nGnRE | XN | - | Shareable | On-chip peripheral address space. |
| 0x60000000- 0x7FFFFFFF | RAM | Normal | - | WBWA RA | Non-shareable | Memory with write-back, write allocate cache attribute for L2 and L3 cache support. |
| 0x80000000- | RAM | Normal | - | WT RA | Non-shareable | Memory with Write-Through cache attribute. |
| 0xA0000000- | Device | Device, nGnRE | XN | - | Shareable | Peripherals accessible to all masters. |
| 0xC0000000- 0xDFFFFFFF | Device | Device, nGnRE | XN | - | Shareable | 1 MB region reserved as the PPB. This supports key resources including the System Control Space, and debug features |
| 0xE0000000- 0xE00FFFFF | System PPB | Device, nGnRnE | XN | - | Shareable | 1 MB region reserved as the PPB. This supports key resources, including the System Control Space, and debug features. |
| 0xE0100000- 0xFFFFFFFF | System Vendor_SYS | Device, nGnRE | XN | - | Shareable | Vendor System Region |

WA - Write-Through.

RA - Read-allocate.

WBWA - Write-back, write-allocate.

XN - Memory with the Execute Never memory attribute.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{RPFG}$ The term boundary is used to indicate the divide between memory regions stated in the system address map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MBRB}$ An access that crosses a boundary is UNPREDICTABLE. This rule also applies to the 0xFFFFFFFF - 0x00000000 boundary.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DNBD}$ An unaligned or multi-word access that crosses a 0.5GB memory region boundary is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B7.2 *The System region of the system address map* on page 242.

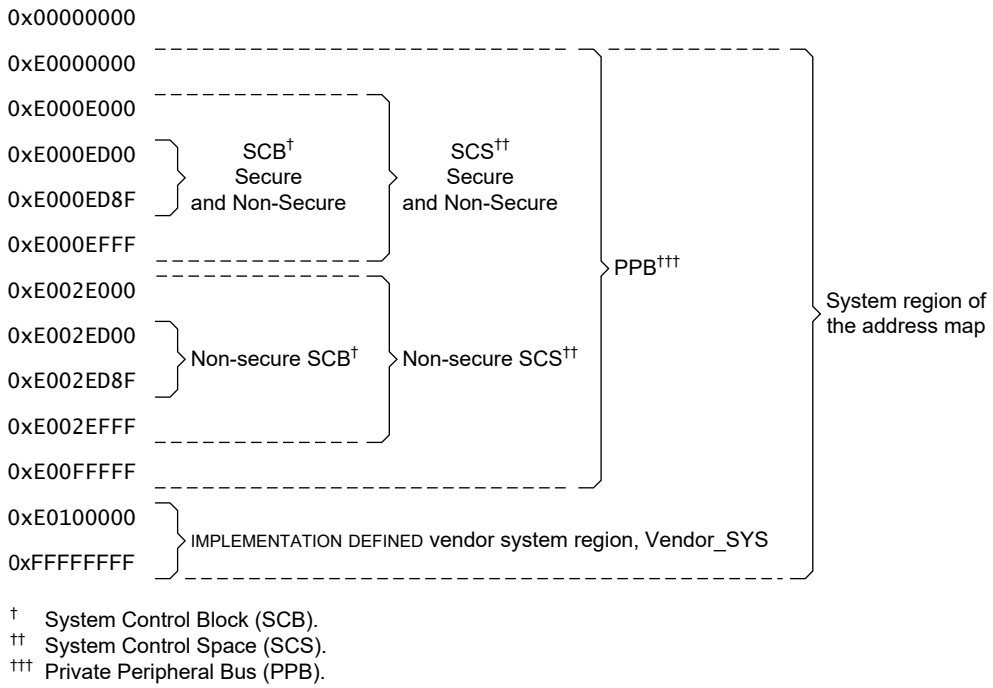
B6.2 *Address space* on page 187.

B6.1 *Memory accesses* on page 186.

B6.24 *Caches* on page 225.

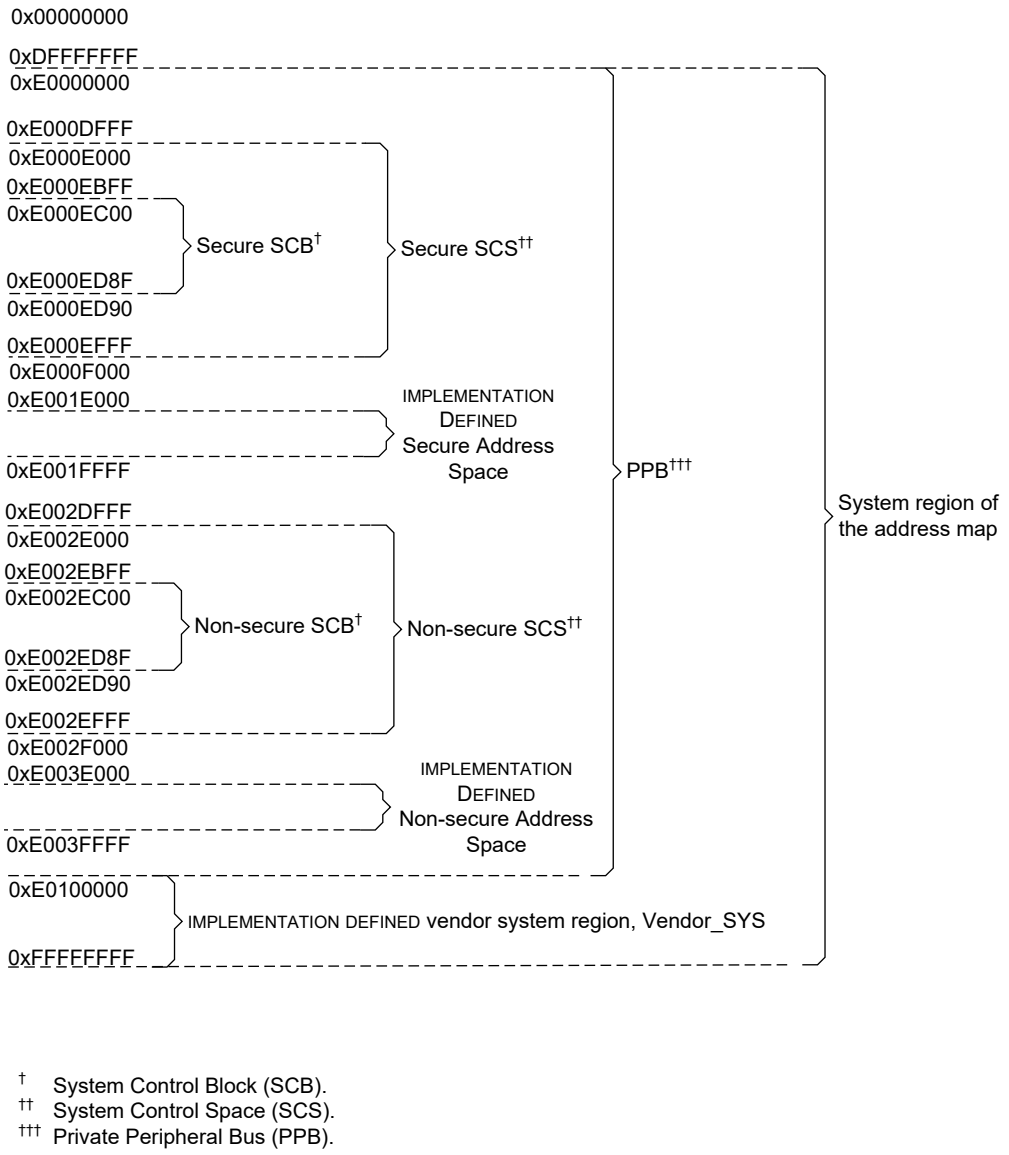
## B7.2 The System region of the system address map

$R_{BDNB}$ The system region of the system address map is as follows:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MHGM}$ The system region of the system address map is as follows:




† System Control Block (SCB).
†† System Control Space (SCS).
††† Private Peripheral Bus (PPB).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{MXRW}$ In a PE without the Security Extension, the Non-secure SCS is RAZ/WI and any unprivileged access to the Non-secure SCS results in a BusFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && !**S**. Note, if !M a HardFault is generated.*

$I_{FWLM}$ Arm recommends that Vendor_SYS is divided as follows:

- `0xE0100000-0xEFFFFFFF` is reserved.
- Vendor resources start at `0xF0000000`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DQQS}$ Unprivileged access to the PPB causes BusFault errors unless otherwise stated. Unprivileged accesses can be enabled to the Software Trigger Interrupt Register in the System Control Space by programming a control bit in the Configuration and Control Register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RJHJ}$ If the exception entry context stacking, exception return context unstacking, lazy floating-point state preservation, or the stacking or unstacking of a FNC_RETURN stack frame, results in an access to an address within the PPB space the behavior of the access is CONSTRAINED UNPREDICTABLE and is one of the following:

- Generates a BusFault.
- Perform the specified access to the PPB space.

This does not apply to the `VLSTM` instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B7.1 *System address map* on page 241.

B7.3 *The System Control Space (SCS)* on page 245.

*STIR, Software Triggered Interrupt Register*.

*CCR, Configuration and Control Register*.

B12.1.2 *Debug resources* on page 278.

## B7.3 The System Control Space (SCS)

$R_{CQVK}$ The System Control Space (SCS) provides registers for control, configuration, and status reporting.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CFPK}$ The Secure view of the NS alias is identical to the Non-secure view of normal addresses unless otherwise stated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{GLNG}$ Privileged accesses to unimplemented registers are RES0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NDML}$ Unprivileged accesses to unimplemented registers will generate a BusFault unless otherwise stated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BMLS}$ The side effects of any access to the SCS that performs a context-altering operation take effect when the access completes. A DSB instruction can be used to guarantee completion of a previous SCS access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WQQB}$ A context synchronization event guarantees that the side effects of a previous SCS access are visible to all instructions in program order following the context synchronization event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B7.2 *The System region of the system address map* on page 242.

System Control Block.

System Control Block (NS alias).

Debug Control Block.

Debug Control Block (NS alias).

*STIR, Software Triggered Interrupt Register*.

*SYST_CSR, SysTick Control and Status Register*.

Chapter B11 *Nested Vectored Interrupt Controller* on page 269.

Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

# Chapter B8
# Synchronization and Semaphores

This chapter specifies the Armv8-M architecture rules for exclusive access instructions and non-blocking synchronization of shared memory. It contains the following sections:

# B8.1 Exclusive access instructions

$R_{LQDX}$ Armv8 provides non-blocking synchronization of shared memory, using synchronization primitives for accesses to both Normal and Device memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RGCP}$ The synchronization primitives and associated instructions are as follows:

| Function | | T32 instruction |
|---|---|---|
| Load-Exclusive | | |
| | Byte | LDREXB, LDAEXB |
| | Halfword | LDREXH, LDAEXH |
| | Word | LDREX, LDAEX |
| Store-Exclusive | | |
| | Byte | STREXB, STLEXB |
| | Halfword | STREXH, STLEXH |
| | Word | STREX, STLEX |
| Clear-Exclusive | | |
| | | CLREX |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MWFP}$ A Load-Exclusive instruction performs a load from memory, and:

- The executing PE marks the memory address for exclusive access.
- The local monitor of the executing PE transitions to the Exclusive Access state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JHMH}$ The size of the marked memory block is called the *Exclusives reservation granule* (ERG), and is an IMPLEMENTATION DEFINED value that is of a power of 2 size, in the range 4 - 512 words.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MTTN}$ A marked block of the ERG is created by ignoring the least significant bits of the memory address. A marked address is any address within this marked block.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FMXK}$ In some implementations the CTR identifies the Exclusives reservation granule. Where this is not the case, the Exclusives reservation granule is treated as having the maximum of 512 words.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B8.2 *The local monitors* on page 248.

B8.3 *The global monitor* on page 250.

B8.4 *Exclusive access instructions and the monitors* on page 254.

B8.5 *Load-Exclusive and Store-Exclusive instruction constraints* on page 255.

## B8.2 The local monitors

$R_{QTFP}$ Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NJWC}$ When a PE writes using any instruction other than a Store-Exclusive instruction:

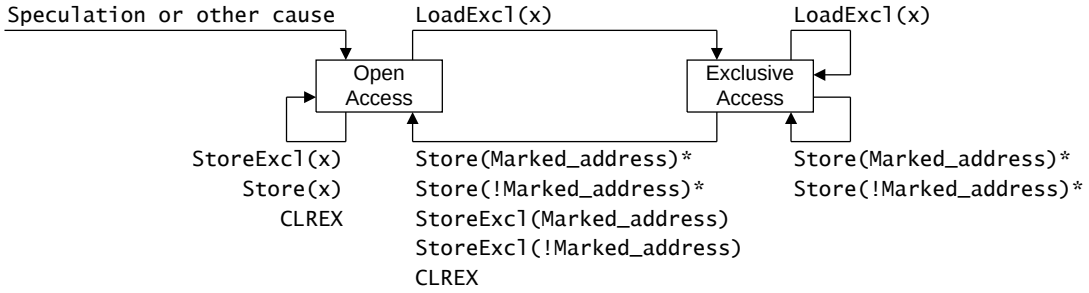
- If the write is to a physical address that is not marked as Exclusive Access by its local monitor and that local monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.
- If the write is to a physical address that is marked as Exclusive Access by its local monitor, it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PFFT}$ It is IMPLEMENTATION DEFINED whether a store to a marked physical address causes a mark in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be marked.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KXNM}$ The state machine for the local monitor is shown here.




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 marked address to the most significant bits of the address `x` used for the operation.

The local monitor only transitions to the Exclusive Access state as the result of the architectural execution of one of the operations shown in the diagram.

Any transition of the local monitor to the Open Access state that is not caused by the architectural execution of an operation shown here does not indefinitely delay forward progress of execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WTHJ}$ The local monitor does not hold any physical address, but instead treats any access as matching the address of the previous Load-Exclusive instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JWQS}$ A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KJQW}$ The architecture does not require a load instruction by another PE that is not a Load-Exclusive instruction to have any effect on the local monitor.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XMML}$ It is IMPLEMENTATION DEFINED whether the transition from Exclusive Access to Open Access state occurs when the `Store` or `StoreExcl` is from another observer.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MRSD}$ The architecture permits a local monitor to transition to the Open Access state as a result of speculation, or from some other cause.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HRHC}$ An exception return clears the local monitor.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B8.4 *Exclusive access instructions and the monitors* on page 254.

## B8.3 The global monitor

$R_{FKFB}$ For each PE in the system, the global monitor:

- Can hold at least one marked block.
- Maintains a state machine for each marked block it can hold.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VDLP}$ For each PE, the architecture only requires global monitor support for a single marked address. Any situation that might benefit from the use of multiple marked addresses on a single PE is CONSTRAINED UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NNDC}$ The global monitor can either reside in a block that is part of the hardware on which the PE executes or exist as a secondary monitor at the memory interfaces.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{XTLH}$ The IMPLEMENTATION DEFINED aspects of the monitors mean that the global monitor and the local monitor can be combined into a single unit, provided that the unit performs the global monitor and the local monitor functions defined in this manual.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{KDWM}$ For shareable memory locations, in some implementations and for some memory types, the properties of the global monitor require functionality outside the PE. Some system implementations might not implement this functionality for all locations of memory. In particular, this can apply to:

- Any type of memory in the system implementation that does not support hardware cache coherency.
- Non-cacheable memory, or memory treated as Non-cacheable, in an implementation that does support hardware cache coherency.

In such a system, it is defined by the system:

- Whether the global monitor is implemented.
- If the global monitor is implemented, which address ranges or memory types it monitors.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{QJNL}$ The only memory types for which it is architecturally guaranteed that a global exclusive monitor is implemented are:

- Inner Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hint and not transient.
- Outer Shareable, Inner Write-Back, Outer Write-Back Normal memory with Read allocation hint and Write allocation hints and not transient.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HBKJ}$ The set of memory types that support atomic instructions includes all of the memory types for which a global monitor is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HLHS}$ If the global monitor is not implemented for an address range or memory type, then performing a Load-Exclusive/Store-Exclusive instruction to such a location, in the absence of any other fault, has one or more of the following effects:

- The instruction generates BusFault.
- The instruction generates a DACCVIOL MemManage fault.

- The instruction is treated as a NOP.
- The Load-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The Store-Exclusive instruction is treated as if it were accessing a Non-shareable location, but the state of the local monitor becomes UNKNOWN.
- The value held in the result register of the Store-Exclusive instruction becomes UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a MemManage Fault requires M && MPU, a BusFault requires M.*

$R_{FQRT}$ For write transactions generated by non-PE observers that do not implement exclusive accesses or other atomic access mechanisms, the effect that writes have on the global monitor and the local monitor that are used by an Arm PE is IMPLEMENTATION DEFINED. The writes might not clear the global monitors of other PEs for:

- Some address ranges.
- Some memory types.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### B8.3.1 Load-Exclusive and Store-Exclusive

$R_{RXVB}$ The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GXLF}$ The architecture does not require a load instruction by another PE, that is not a Load-Exclusive instruction, to have any effect on the global monitor.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MPKM}$ A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MFGC}$ A Store-Exclusive instruction performs a conditional store to memory:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NNMG}$ In a shared memory system, the global monitor implements a separate state machine for each PE in the system. The state machine for accesses to shareable memory by PE(n) can respond to all the shareable memory accesses visible to it.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NWWH}$ Whenever the global monitor state for a PE changes from Exclusive access to Open access, an event is generated and held in the Event register for that PE. This register is used by the Wait for Event mechanism.

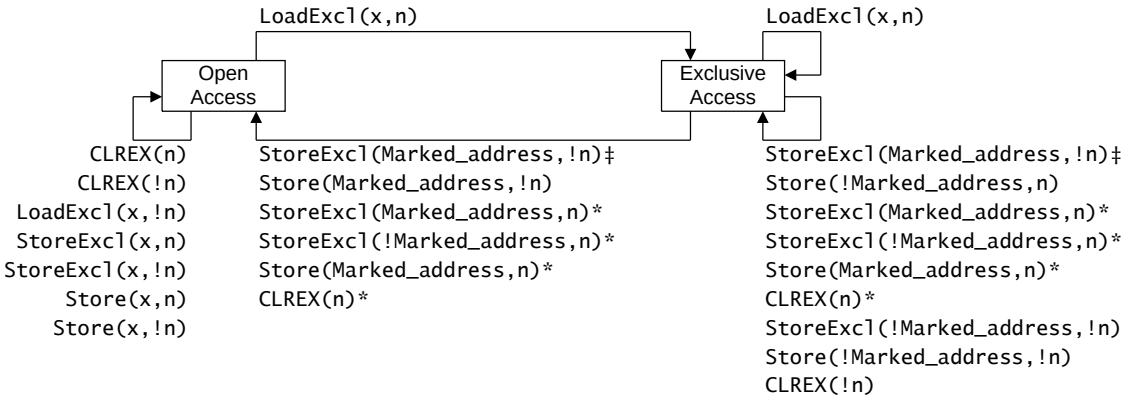
*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### B8.3.2 Load-Exclusive and Store-Exclusive in Shareable memory

$R_{HKQT}$ A Load-Exclusive instruction from shareable memory performs a load from memory, and causes the physical address of the access to be marked as exclusive access for the requesting PE. This access can also cause the exclusive access mark to be removed from any other physical address that has been marked by the requesting PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GDMD}$ The state machine for PE(n) in a global monitor is as follows.




‡StoreExcl(Marked_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 marked address to the most significant bits of the address x used for the operation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RGFK}$ Whether a Store-Exclusive instruction successfully updates memory or not depends on whether the address accessed matches the marked shareable memory address for the PE issuing the Store-Exclusive instruction, and whether the local monitor and the global monitor are in the exclusive state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QVWF}$ When the global monitor is in the Exclusive Access state, it is IMPLEMENTATION DEFINED whether a `CLREX` instruction causes the global monitor to transition from Exclusive Access to Open Access state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DLMP}$ A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BSGB}$ It is IMPLEMENTATION DEFINED:

- Whether a modification to a Non-shareable memory location can cause a global monitor to transition from Exclusive Access to Open Access state.
- Whether a Load-Exclusive instruction to a Non-shareable memory location can cause a global monitor to transition from Open Access to Exclusive Access state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B8.4 *Exclusive access instructions and the monitors* on page 254.

## B8.4 Exclusive access instructions and the monitors

$R_{VXWN}$ The Store-Exclusive instruction defines the register to which the status value of the monitors is returned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DTRN}$ A Store-Exclusive instruction performs a conditional store to memory that depends on the state of the local monitor:

- **If the local monitor is in the Exclusive Access state**:
  - If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
  - A status value is returned to a register:
    - If the store took place the status value is 0.
    - Otherwise, the status value is 1.
  - The local monitor of the executing PE transitions to the Open Access state.
- **If the local monitor is in the Open Access state**:
  - No store takes place.
  - A status value of 1 is returned to a register.
  - The local monitor remains in the Open Access state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DFNB}$ A Store-Exclusive instruction performs a store to Shareable memory that depends on the state of both the local monitor and the global monitor:

- **If both the local monitor and the global monitor are in the Exclusive Access state**:
  - If the address of the Store-Exclusive instruction is the same as the address that has been marked in the monitor by an earlier Load-Exclusive instruction, then the store occurs. Otherwise, it is IMPLEMENTATION DEFINED whether the store occurs.
  - A status value is returned to a register:
    - If the store took place the status value is 0.
    - Otherwise, the status value is 1.
  - The local monitor of the executing PE transitions to the Open Access state.
- **If either the local monitor or the global monitor is in the Open Access state:**
  - No store takes place.
  - A status value of 1 is returned to a register.
  - The local monitor of the executing PE transitions to the Open Access state.
  - The global monitor that is associated with the executing PE transitions to the Open Access state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B8.2 *The local monitors* on page 248.

B8.3 *The global monitor* on page 250.

## B8.5 Load-Exclusive and Store-Exclusive instruction constraints

$I_{RTHW}$ The Load-Exclusive and Store-Exclusive instructions are intended to work together as a pair, for example a `LDREX/STREX` pair or a `LDREXB/STREXB` pair.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BHPN}$ The architecture does not require an address or size check as part of the `IsExclusiveLocal()` function.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LHLG}$ If two `StoreExcl` instructions are executed without an intervening `LoadExcl` instruction the second `StoreExcl` instruction returns a status value of 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DVRQ}$ The architecture does not require every `LoadExcl` instruction to have a subsequent `StoreExcl` instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JXXS}$ If the transaction size of a `StoreExcl` instruction is different from the preceding `LoadExcl` instruction in the same thread of execution, behavior is a CONSTRAINED UNPREDICTABLE choice of:

- The `StoreExcl` either passes or fails, and the status value returned by the `StoreExcl` is UNKNOWN.
- The block of data of the size of the larger of the transaction sizes used by the `LoadExcl/StoreExcl` pair at the address accessed by the `LoadExcl/StoreExcl` pair, is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GVWN}$ The hardware only ensures that a `LoadExcl/StoreExcl` pair succeeds if the `LoadExcl` and the `StoreExcl` have the same transaction size.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XLSK}$ Forward progress can only be made using `LoadExcl/StoreExcl` loops if, for any `LoadExcl/StoreExcl` loop within a single thread of execution if both of the following are true:

- There are no explicit memory accesses, pre-loads, direct or indirect register writes, cache maintenance instructions, `SVC` instructions, or exception returns between the Load-Exclusive and the Store-Exclusive.
- The following conditions apply between the Store-Exclusive having returned a fail result and the retry of the Load-Exclusive:
  - There are no stores to any location within the same Exclusives reservation granule that the Store-Exclusive is accessing.
  - There are no direct or indirect register writes, other than changes to the flag fields in APSR or FPSCR, caused by data processing or comparison instructions.
  - There are no direct or indirect cache maintenance instructions, `SVC` instructions, or exception returns.

  The exclusive monitor can be cleared at any time without an application-related cause, provided that such clearing is not systematically repeated so as to prevent the forward progress in finite time of at least one of the threads that is accessing the exclusive monitor.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{RFXR}$ Keeping the `LoadExcl` and the `StoreExcl` operations close together in a single thread of execution minimizes the chance of the exclusive monitor state being cleared between the `LoadExcl` instruction and the `StoreExcl` instruction. Therefore, for best performance, Arm strongly recommends a limit of 128 bytes between `LoadExcl` and `StoreExcl` instructions in a single thread of execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PKQF}$ The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{PGGN}$ For performance reasons, Arm recommends that objects that are accessed by exclusive accesses are separated by the size of the exclusive reservations granule.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XPDN}$ After taking a BusFault or a MemManage fault, the state of the exclusive monitors is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{FCRN}$ For the memory location accessed by a `LoadExcl/StoreExcl` pair, if the memory attributes for a `StoreExcl` instruction are different from the memory attributes for the preceding `LoadExcl` instruction in the same thread of execution, behavior is CONSTRAINED UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DMJW}$ The effect of a data or unified cache invalidate, clean, or clean and invalidate instruction on a local exclusive monitor or a global exclusive monitor that is in the Exclusive Access state is CONSTRAINED UNPREDICTABLE, and the instruction might clear the monitor, or it might leave it in the Exclusive Access state. For address-based maintenance instructions, this also applies to the monitors of other PEs in the same Shareability domain as the PE executing the cache maintenance instruction, as determined by the Shareability domain of the address being maintained.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{MDHL}$ Arm strongly recommends that implementations ensure that the use of such maintenance instructions by a PE in the Non-secure state cannot cause a denial of service on a PE in the Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RRTJ}$ In the event of repeatedly-contending `LoadExcl`/`StoreExcl` instruction sequences from multiple PEs, an implementation ensures that forward progress is made by at least one PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

# Chapter B9
# The Armv8-M Protected Memory System Architecture

This chapter specifies the Armv8-M *Protected Memory System Architecture* (PMSAv-8) rules, and in particular the rules for the optional *Memory Protection Unit*(MPU) and the optional *Security Attribution Unit* (SAU). It contains the following sections:

## B9.1 Memory Protection Unit

$R_{HPNK}$ In an implementation that includes the Protected Memory System Architecture (PMSA), system address space is protected by a Memory Protection Unit (MPU).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{TBPJ}$ PMSAv8-M only supports a unified memory model. All enabled regions support instruction and data accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{HBNG}$ Memory attributes are determined from the default system address map or by using an MPU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{BXCN}$ MPU support in Armv8-M is optional.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{MCCL}$ The default memory map can be configured to provide a background region for privileged accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{JVJC}$ When the MPU is disabled or not present, accesses use memory attributes from the default system address map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!MPU**.*

$R_{KLHL}$ If the MPU is enabled, attributes for memory accesses that hit in a single region are provided by the hit region.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{DBBM}$ The MPU divides the memory into regions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{JVCN}$ An individual MPU region is defined by:

```
Address >= MPU_RBAR.BASE:'00000' && Address <= MPU_RLAR.LIMIT:'11111'
```

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{MNDS}$ The number of supported MPU regions is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S && MPU**.*

$I_{WTCL}$ Because the MPU_TYPE register is banked, an implementation can have a different number of MPU regions, including no MPU regions, for each Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{XGFK}$ All MPU regions are aligned to a multiple of 32 bytes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{BPGB}$ The PE can fetch and execute instructions from each MPU region according to the value of MPU_RBAR.XN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{NBPN}$ Accesses to the following region of memory `0xE0000000-0xE00FFFFF`, the *Private Peripheral Bus* (PPB) always use memory attributes from the default system address map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{ZLHD}$ Unless otherwise stated, all load, store, and instruction fetch transactions are subject to an MPU check.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{DNXT}$ If MPU_CTRL.ENABLE is zero, MPU checks are carried out against the default address map and not against any defined MPU regions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$I_{HSCD}$ The MPU check is one of a number of checks carried out on any load, store or instruction fetch transaction including alignment, security attribution checks, and a check for any BusFaults.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{VHHL}$ Exception vector reads from the Vector Address Table always use the default address map and are not subject to an MPU check.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{VWXJ}$ If MPU_CTRL.HFNMIENA is set to 0, any load, store or instruction fetch transaction where the requested execution priority is negative will use the Default Address Map for MPU checks.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{TGQD}$ Any load, store or instruction fetch transactions to the PPB, within the range `0xE0000000-0xE00FFFFF`, are not subject to an MPU check but are checked against the default address map. Instruction fetches to this region generate an XN MemManage fault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{LLLP}$ Any MPU region lookup performed for a load, store or instruction fetch transaction will generate a precise MemManage Fault if any of the following is true:

- The address accessed by the load, store or instruction fetch transaction matches more than one MPU region.
- The load, store or instruction fetch transaction does not match all of the access conditions for the MPU region being accessed.
- The load, store or instruction fetch transaction matches a background region or the default memory map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{KDJG}$ The MPU is restricted in how it can change the default memory map attributes associated with System space, that is, for addresses in the region `0xE0100000-0xFFFFFFFF`. Unless otherwise stated, system space is always XN (Execute Never) and it is always Device-nGnR. If the MPU maps this to a type other than Device-nGnRnE, it is UNKNOWN whether the region is treated as Device-nGnRE or as Device-nGnRnE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{KMTF}$ Unless otherwise stated for data accesses, the MPU memory attribution and privilege checking uses the configuration registers that correspond to the current Security state of the PE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU && S** .*

$R_{RLBR}$ For instruction fetches, the MPU memory attribution and privilege checking uses the configuration registers associated with the security of the target address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{PLJG}$ Setting MPU_CTRL.HFNMIENA. to zero disables the MPU if the requested priority for the handler of the HardFault, NMI and exceptions that the MPU is associated with is negative.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

$R_{RJJL}$ When MPU_RLAR.PXN == 1, a MemManage fault is generated if the PE is executing in a privileged mode and attempts to execute an instruction from the corresponding memory region.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MPU** && **M** && **PXN**.*

$R_{MKJC}$ Setting the MPU_RNR.REGION field to a value that does not correspond to an implemented memory region is CONSTRAINED UNPREDICTABLE as follows:

- Any subsequent read of MPU_RNR.REGION returns an UNKNOWN value.
- Any read of a register that is in an unimplemented region returns an UNKNOWN value.
- Any write to a register indirected by MPU_RNR.REGION causes all state that is indirected by that register to become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **MPU**.*

See also:

B7.1 *System address map* on page 241.

B6.7 *Access rights* on page 195.

B6.17 *Device memory attributes* on page 213.

B6.19 *Shareability attributes* on page 218.

B6.20 *Memory access restrictions* on page 219.

B6.21 *Mismatched memory attributes* on page 220.

B6.22 *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page 222.

B6.23 *Load-Acquire and Store-Release accesses to memory* on page 223.

*MPU_CTRL, MPU Control Register*.

*TT_RESP, Test Target Response Payload*.

## B9.2 Security attribution

$I_{SBSJ}$ The Secure Attribution Unit and the Implementation Defined Attribution Unit are collectively referred to as the Attribution Unit (AU).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{JGHS}$ The Security Extension defines three levels of memory security attribution. In ascending order of security, these are:

1. Non-secure.
2. Secure and Non-secure callable.
3. Secure and not Non-secure callable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{RPKG}$ The following units can provide security attribution information:

- A *Security attribution unit* (SAU) inside the PE.
- An IMPLEMENTATION DEFINED *attribution unit* (IDAU) external to the PE. The presence of such a unit is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{MGXN}$ The attribution information from the SAU is used unless the IDAU specifies attributes with a higher security, in which case the IDAU attributes override the SAU attributes. This rule does not apply to architecturally defined ranges exempt from memory attribution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{NJGR}$ An *attribution unit* (AU) violation is defined as being a violation raised by either the SAU or the IDAU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{QGVS}$ All boundaries between address ranges with different security attributes are aligned to 32-byte boundaries.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{BLJT}$ The behavior of the following address ranges is fixed, so they are exempt from memory attribution by both the SAU and IDAU:

**0xF0000000 – 0xFFFFFFFF**

If the PE implements the Security Extension, this memory range is always marked as Secure and not Non-secure callable for instruction fetches.

If the Security Extension is not present, this range is marked as Non-secure.

**Ranges exempt from checking security violation**

The following address ranges are marked with the Security state indicated by NS-Req, that is, the current state of the PE for non-debug accesses. This marking sets the NS-Attr to NS-Req:

`0xE0000000` - `0xE0002FFF`: ITM, DWT, FPB.

`0xE000E000` - `0xE000EFFF`: SCS range.

`0xE002E000` - `0xE002EFFF`: SCS NS alias range.

`0xE0040000` - `0xE0041FFF`: TPIU, ETM.

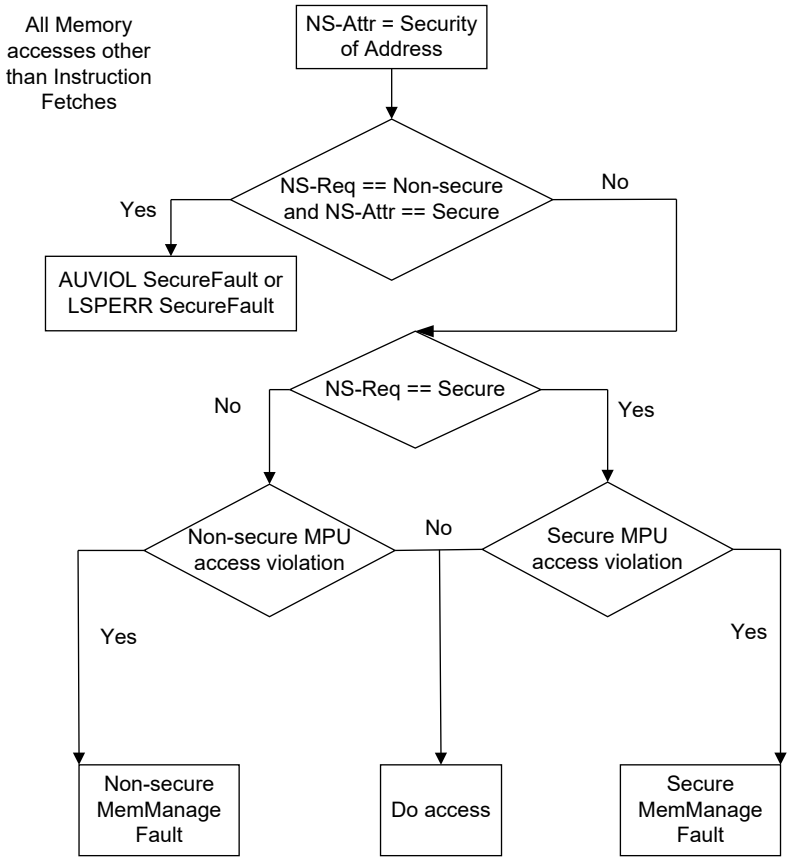
`0xE00FF000` - `0xE00FFFFF`: ROM table.

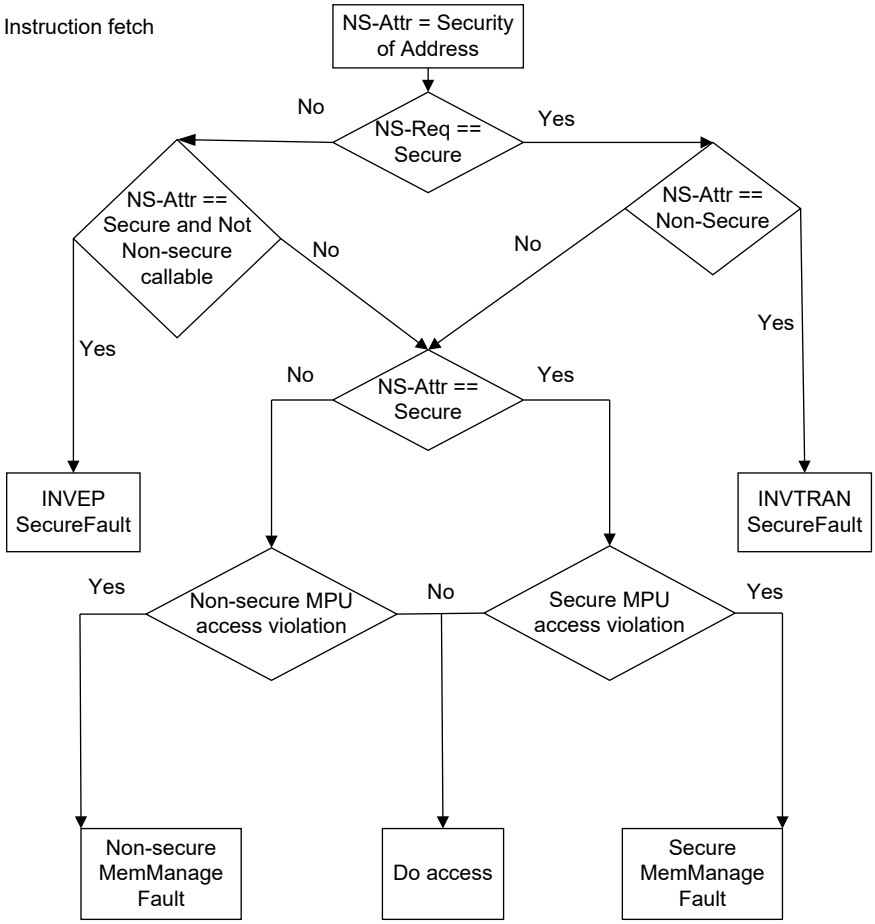
`0xE0000000` - `0xEFFFFFFF`: for instruction fetch only.

Additional address ranges specified by the IDAU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**. Note, some address ranges require DB.*

$I_{VPWL}$ The Security attribution and MPU check sequence, for all data accesses which are not instruction fetches and accesses for instruction fetches are shown in the following diagrams.






If the memory transaction is lazy state preservation then NS-req is determined by FPCCR.S. If the memory transaction is an exception vector fetch access NS-Req is set to the Security state of the exception. Unless otherwise specified, in all other cases it is the current Security state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{FTFR}$ If the memory access is caused by the execution of an `SG` instruction the load from the stack is undertaken with the NS-Req set to Secure even though the `SG` instruction was executed from the Non-secure state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **S**.*

See also:

B9.3 *Security attribution unit (SAU)* on page 264.

B9.4 *IMPLEMENTATION DEFINED Attribution Unit (IDAU)* on page 265.

## B9.3 Security attribution unit (SAU)

$R_{VFLR}$ The SAU configuration defines an IMPLEMENTATION DEFINED number of memory regions. The number of regions is indicated by SAU_TYPE.SREGION.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$I_{PPLK}$ The memory regions defined by the SAU configuration are referred to as SAU_REGIONn, where n is a number from 0 - (SAU_TYPE.SREGION-1).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{RVFP}$ The SAU region configuration fields can only be accessed indirectly using the window registers, SAU_RNR shown in the following table.

| SAU region configuration field | Associated window register field |
|---|---|
| SAU_REGIONn.ENABLE | SAU_RLAR.ENABLE |
| SAU_REGIONn.NSC | SAU_RLAR.NSC |
| SAU_REGIONn.BADDR | SAU_RBAR.BADDR |
| SAU_REGIONn.LADDR | SAU_RLAR.LADDR |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{KRSC}$ When the SAU is enabled, an address is defined as matching a region in the SAU if the following is true:

```
Address >= SAU_REGIONn.BADDR: '00000' && Address <= SAU_REGIONn.LADDR: '11111'.
```

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{MPJC}$ Memory is marked as Secure by default. However, if the address matches a region with SAU_REGIONn.ENABLE set to 1 and SAU_REGIONn.NSC set to 0, then memory is marked as Non-secure.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{WGDK}$ An address that matches multiple SAU regions is marked as Secure and not Not-secure callable regardless of the attributes specified by the regions that matched the address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{GVFQ}$ When the SAU is not enabled:

- Addresses are not checked against the SAU regions.
- The attribution of the address space is determined by the SAU_CTRL.ALLNS field.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{MBJN}$ To permit lockdown of the SAU configuration, it is IMPLEMENTATION DEFINED whether SAU_RLAR, SAU_RBAR, SAU_CTRL, and SAU_RNR are writable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{BBCT}$ Setting the SAU_RNR.REGION field to a value that does not correspond to an implemented memory region is CONSTRAINED UNPREDICTABLE as follows:

- Any subsequent read of SAU_RNR.REGION returns an UNKNOWN value.
- Any read of a register that is in an unimplemented region returns an UNKNOWN value.
- Any write to a register indirected by SAU_RNR.REGION causes all state that is indirected by that register to become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

## B9.4 IMPLEMENTATION DEFINED Attribution Unit (IDAU)

$R_{MVCM}$ The IDAU can provide the following Security attribution information for an address:

- Security attribution exempt. This specifies that the address is exempt from security attribution. This information is combined with the address ranges that are architecturally required to be exempt from attribution.
- Non-secure. This specifies if the address is Secure or Non-secure.
- Non-secure callable. This specifies if code at the address can be called from Non-secure state. This attribute is only valid if the address is marked as Secure.
- Region number. This is the region number that matches the address, and is only used by the `TT` instruction.
- Region number valid. This specifies that the region number is valid. This field has no effect on the attribution of the address, and is only used by the `TT` instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

`TT, TTT, TTA, TTAT`.

B9.2 *Security attribution* on page 261.

# Chapter B10
# The System Timer, SysTick

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

## B10.1 The system timer, SysTick

$R_{BQRG}$ In a PE without the Main Extension and without the Security Extensions, either:

- No system timers are implemented.
- One system timer, SysTick, is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M && !S**.*

$R_{PDDL}$ In a PE without the Main Extension but with the Security Extension, one of the following is true:

- No system timers are implemented.
- One system timer, SysTick, is implemented. ICSR.STTNS determines which Security state owns the SysTick.
- Two system timers are implemented:
  - SysTick, Secure instance.
  - SysTick, Non-secure instance.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M && S**.*

$R_{CNTG}$ In a PE with the Main Extension but without the Security Extension, one system timer, SysTick, is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M && ST && !S**.*

$R_{XPCW}$ In a PE with the Main and Security Extensions, two system timers are implemented:

- SysTick, Secure instance.
- SysTick, Non-secure instance.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M && S && ST**.*

$I_{DXSQ}$ There are the following SysTick registers:

- SysTick Control and Status Register (SYST_CSR).
- SysTick Reload Value Register (SYST_RVR).
- SysTick Current Value Register (SYST_CVR).
- SysTick Calibration Value Register (SYST_CALIB).

In a PE with the Security Extension and a SysTick instance dedicated to each Security state, these registers are banked.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$I_{VHDT}$ Each implemented SysTick is a 24-bit decrementing, wrap-on-zero, clear-on-write counter:

- When enabled, the counter counts down from the value in SYST_CVR, SYST_CVR. When it reaches zero, SYST_CVR is reloaded with the value held in SYST_RVR on the next clock edge.
- Reading SYST_CVR returns the value of the counter at the time of the read access.
- When the counter reaches zero, it sets SYST_CSR.COUNTFLAG to 1. Reading SYST_CSR.COUNTFLAG clears it to 0.
- A write to SYST_CVR clears both SYST_CVR and SYST_CSR.COUNTFLAG to 0. SYST_CVR is then reloaded with the value held in SYST_RVR on the next clock edge.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$R_{TLGK}$ Writing the value zero to SYST_RVR disables the SysTick on the next wrap-on-zero. The value zero is held by the counter after the wrap. This is true even when SYST_CSR.ENABLE is 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$R_{TTFT}$ A write to SYST_CVR does not cause a SysTick exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$I_{VDJQ}$ Setting SYST_CSR.TICKINT to 1 causes the SysTick exception to become pending on the SysTick reaching zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$I_{PPGV}$ Arm recommends that before enabling a SysTick by SYST_CSR.ENABLE, software writes the required counter value to the SYST_RVR, and then writes to the SYST_CVR to clear the SYST_CVR to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$I_{MMRQ}$ Software can optionally use SYST_CALIB.TENMS to scale the counter to other clock rates within the dynamic range of the counter.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

$R_{QSKV}$ When the PE is halted in Debug state, any implemented SysTicks do not decrement.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST && Halting debug**.*

$I_{RWFQ}$ Each implemented SysTick is clocked by a reference clock, either the PE clock or an external system clock. It is IMPLEMENTATION DEFINED which clock is used as the external reference clock. Arm recommends that if an external system clock is used, the relationship between the PE clock and the external clock is documented, so that system timings can be calculated taking into account metastability, clock skew, and jitter.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ST**.*

# Chapter B11
# Nested Vectored Interrupt Controller

This chapter specifies the Armv8-M *Nested Vectored Interrupt Controller* (NVIC) rules. It contains the following sections:

## B11.1 NVIC definition

$R_{XJJQ}$ An Armv8-M PE includes an integral interrupt controller.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WQHG}$ The Interrupt Controller Type Register (ICTR) defines the number of external interrupt lines that are supported.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

*ICTR, Interrupt Controller Type Register*.

## B11.2 NVIC operation

$R_{SNVK}$ It is IMPLEMENTATION DEFINED which NVIC interrupts are implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SGCR}$ When a particular NVIC interrupt line is not implemented, the registers that are associated with it are reserved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CCVJ}$ Only an interrupt that is both pending and enabled with sufficient priority can preempt PE execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CVJS}$ The following events on the input associated with an interrupt cause the pending state associated with the interrupt to become set:

- The input is HIGH while the active state associated with the interrupt is clear.
- The input transitions from LOW to HIGH while the active state associated with the interrupt is set.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{WTFS}$ The Armv8-M interrupt behavior provides compatibility with both active-high level-sensitive and pulse-sensitive interrupt signaling:

- For level-sensitive interrupts, the associated exception handler runs one time for each occurrence as long as the level is cleared before the exception handler returns. If the level of the input is HIGH after the exception handler returns, the exception will be pended again.
- For pulse-sensitive interrupts, the associated exception handler runs one time only, regardless of the number of pulses that the NVIC sees before the exception handler is entered. If a pulse occurs after the exception handler has been entered, the exception will be pended again.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{HVQQ}$ For some implementations, pulse-sensitive interrupt signals are held long enough to ensure that the PE can sample them reliably.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QKFW}$ All NVIC interrupts have a programmable priority value and an associated exception number.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XNQW}$ NVIC interrupts can be enabled and disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit field.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WGDJ}$ An implementation can hard-wire interrupt enable bits to zero if the associated interrupt line does not exist.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RSDJ}$ An implementation can hard-wire interrupt enable bits to one if the associated interrupt line cannot be disabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NRJV}$ It is IMPLEMENTATION DEFINED for each NVIC interrupt line supported whether an NVIC interrupt supports either or both setting and clearing of the associated pending state under software control.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.13 *Priority model* on page 91.

Nested Vectored Interrupt Controller Block.

Nested Vectored Interrupt Controller Block(NS alias).

# Chapter B12
# **Debug**

This chapter specifies the Armv8-M debug rules. It contains the following sections:

## B12.1 Debug feature overview

$R_{WXRJ}$ The debug configuration of an implementation is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FMQF}$ The following table sets out the optional features of the Armv8-M debug architecture.

| Feature | Main Extension | Baseline Implementation |
|---|---|---|
| **DebugMonitor exception** | Always implemented | Never implemented |
| **Halting debug** | Optional | Optional |
| **EDBGRQ**<br>*External Halt signal* | Optional | Requires Halting debug |
| **Flash Patch and Breakpoint unit - FPB** | Optional | Requires Halting debug |
| **Data Watchpoint and Trace Unit - DWT** | | |
| *Debug functionality - DWT-D* | Optional | Requires Halting debug |
| *Trace functionality - DWT-T* | Requires ITM and Debug functionality | Never implemented |
| **Instrumenation Trace Macrocell - ITM** | Optional | Never implemented |
| **Cross Trigger Interface - CTI** | Requires ETM or Halting Debug | Requires ETM or Halting Debug |
| **Trace Port Interface Unit - TPIU** | Requires ITM or ETM | Requires ETM |
| **Embedded Trace Marcocell - ETM** | Optional | Optional |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GPKJ}$ The following table sets out the optional features of the Armv8-M debug architecture.

| Feature | Main Extension | Baseline Implementation |
|---|---|---|
| **DebugMonitor exception** | Always implemented | Never implemented |
| **Halting debug** | Optional | Optional |
| **EDBGRQ**<br>*External Halt signal* | Optional | Requires Halting debug |
| **Flash Patch and Breakpoint unit - FPB** | Optional | Requires Halting debug |
| **Data Watchpoint and Trace Unit - DWT** | | |
| *Debug functionality - DWT-D* | Optional | Requires Halting debug |
| *Trace functionality - DWT-T* | Requires ITM and Debug functionality | Never implemented |
| **Instrumenation Trace Macrocell - ITM** | Optional | Never implemented |
| **Cross Trigger Interface - CTI** | Requires ETM or Halting Debug | Requires ETM or Halting Debug |
| **Trace Port Interface Unit - TPIU** | Requires ITM or ETM | Requires ETM |
| **Embedded Trace Marcocell - ETM** | Optional | Optional |
| **Performance Monitors Unit - PMU** | Optional | Never implemented |
| **Unprivileged Debug Extension - UDE** | Optional | Never implemented |
| **DSP Debug Extension - DSPDE** | Optional | Never implemented |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**. Note, CTI requires Halting debug or ETM.*

$R_{FHRN}$ The following optional debug components are not part of the Armv8-M architecture:

- The *Cross-Trigger Interface* (CTI).
- The CoreSight basic trace router (MTB).
- The Embedded Trace Macrocell (ETM).

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, CTI requires Halting debug or ETM.*

$I_{SFSG}$ The recommended Debug implementation levels are:

- Minimum.
- Basic.
- Comprehensive.
- Program trace.

**Minimum**

In an implementation that includes the Main Extension, the minimum level contains support for the DebugMonitor exception, including:

- The BKPT instruction.
- DEMCR Monitor debug features.
- Monitor entry from External debug requests.
- DFSR.

DHCSR, DCRSR, DCRDR, and the Halting debug features in DFSR and DEMCR are RES0.
ID_DFR0 is RAZ.

In an implementation that does not include the Main Extension there is no debug support.

DFSR, DHCSR, DCRSR, DCRDR, and DEMCR are RES0.
ID_DFR0 is RAZ.

**Basic**

In an implementation that includes the Main Extension, the basic level adds support for Halting debug with:

- A Debug Access Port and ROM table.
- DHCSR, DCRSR, DCRDR, and the Halting debug features in DEMCR are implemented.
- FPB with at least two breakpoints.
- DWT with at least:
  - One watchpoint that supports instruction, data address, and data value matching.
  - DWT_PCSR.
- Optional support for a CTI in a multiprocessor system.

Support for the basic implementation is identified by ID_DFR0.

In an implementation that does not include the Main Extension, the basic level adds support for Halting debug with:

- A Debug Access Port and ROM table.
- SHCSR, DFSR, DHCSR, DCRSR, DCRDR, and DEMCR are implemented. Access for the PE is IMPDEF.
- FPB with at least two breakpoints.
- DWT with at least:
  - One watchpoint that supports instruction, data address, and data value matching.
  - DWT_PCSR.
- Optional support for a CTI in a multiprocessor system.

Support for the basic implementation is identified by ID_DFR0.

**Comprehensive**

In an implementation that includes the Main Extension, the comprehensive level adds basic trace support with:

- An ITM.
- DWT with:
  - Trace support.
  - Profiling support.
  - Cycle counter.
- TPIU.

In an implementation that does not include the Main Extension, there is no support for the comprehensive level.

**Program trace**

In an implementation that includes the Main Extension, Program trace adds support for ETMs.

In an implementation that does not include the Main Extension, Program trace adds supports for ETMs and TPIUs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{ZNHD}$ An Armv8.1-M implementation introduces further optional debug extensions:

**Performance Monitoring**

In an implementation that includes the Main Extension, Performance Monitoring adds support for PMU.

In an implementation that does not include the Main Extension, there is no support for Performance Monitoring.

**Unprivilged Debug Extension**

In an implementation that includes the Main Extension, the Unprivileged Debug Extension adds support for UDE.

In an implementation that does not include the Main Extension, there is no support for the Unprivileged Debug Extension.

**DSP Debug Extension**

In an implementation that includes the Main Extension, the DSP Debug Extension adds support for DSPDE.

In an implementation that does not include the Main Extension, there is no support for the DSP Debug Extension.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**.*

$R_{KCYJ}$ Armv8.1-M introduces implicit branching for BF instructions, and this behavior might induce implicit changes in the program flow. In the case where a breakpoint or watchpoint is set on the instruction after the BF branch point and the implicit branch is taken, the breakpoint or watchpoint will have no effect.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

$R_{FBRP}$ It is UNKNOWN whether a Watchpoint or a Breakpoint that is targeted at an LE or LETP instruction will have any effect.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB** && (**DWT-D** || **FPB**).*

$I_{SWHC}$ If the debugger expects predictable control flow, then Arm recommends that the implicit branches are disabled and that the associated cache is cleared.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **LOB**.*

See also:

B12.1.1 *Debug mechanisms* .

Halting debug.

DebugMonitor exception.

B12.4.4 *Breakpoint instructions* on page 315.

B13.1 *Instrumentation Trace Macrocell* on page 325.

B13.2 *Data Watchpoint and Trace unit* on page 334.

B13.3 *Embedded Trace Macrocell* on page 356.

B13.4 *Trace Port Interface Unit* on page 357.

B13.5 *Flash Patch and Breakpoint unit* on page 359.

*DEMCR, Debug Exception and Monitor Control Register*.

*DFSR, Debug Fault Status Register*.

*DHCSR, Debug Halting Control and Status Register*.

*DCRDR, Debug Core Register Data Register*.

*DCRSR, Debug Core Register Select Register*.

*ID_DFR0, Debug Feature Register*.

*DWT_PCSR, DWT Program Control Sample Register* .

### B12.1.1 Debug mechanisms

$R_{HWCH}$ Armv8-M supports a range of invasive and non-invasive debug mechanisms. The *invasive debug mechanisms* are:

- The ability to halt the PE. This provides a run-stop debug model.
- Debugging code using the DebugMonitor exception. This provides less intrusive debug than halting the PE.

The *non-invasive debug techniques* are:

- Generating application trace by writing to the *Instrumentation Trace Macrocell* (ITM), causing a low level of intrusion.
- Non-intrusive program trace and profiling.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**. Note, M is required for the DebugMonitor exception and ITM.*

$I_{LBLF}$ When the PE is halted, it is in *Debug state*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{SXVR}$ When the PE is not halted, it is in *Non-debug state*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{DMQP}$ The *Unprivileged Debug Extension*, UDE, allows for a finer-grained control of debug access to the PE. UDE allows conditional debug capabilities when the PE is in an unprivileged mode, including support for the DebugMonitor

exception and the optional Halting debug and non-invasive debug.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE**.*

See also:

B12.2 *Accessing debug features* on page 281.

### B12.1.2 Debug resources

$R_{TZVG}$ In the system address map, debug resources are in the *Private Peripheral Bus* (PPB) region.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FBHD}$ Except for the resources in the SCS, each debug component occupies a fixed 4KB address region.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{WXTK}$ The debug resources in the SCS are:

- The *Debug Control Block* (DCB).
- Debug controls in the *System Control Block* (SCB).

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{KKBT}$ If the Main Extension is implemented, then support for DebugMonitor is implemented. If the Main Extension is not implemented, then DebugMonitor is not supported.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**. Note, M is required for DebugMonitor exception.*

$R_{VMGD}$ ROM table entries identify which optional debug components are implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{RNXK}$ The addresses of the optional debug resources are:

| Address range | Debug Resource | |
|---|---|---|
| 0xE0000000-0xE0000FFF | *Instrumentation Trace Macrocell* (ITM) | |
| 0xE0001000-0xE0001FFF | *Data Watchpoint and Trace* (DWT) Unit | |
| 0xE0002000-0xE0002FFF | *Flashpatch and Breakpoint* Unit (FPB) | |
| 0xE000E000-0xE000EFFF | Secure SCS | |
| | 0xE000ED00-0xE000ED8F | *Secure System Control Block* (SCB) |
| | 0xE000EDF0-0xE000EEFF | *Secure Debug Control Block* (DCB) |
| 0xE002E000-0xE002EFFF | Non-secure SCS | |
| | 0xE002ED00-0xE002ED8F | *Non-secure System Control Block* (SCB) |
| | 0xE002EDF0-0xE002EEFF | *Non-secure Debug Control Block* (DCB) |
| 0xE0040000-0xE0040FFF | *Trace Port Interface Unit* (TPIU), when not implemented as a shared resource otherwise reserved. | |
| 0xE0041000-0xE0041FFF | *Embedded Trace Macrocell* (ETM) | |
| 0xE0042000-0xE00FEFFF- | IMPLEMENTATION DEFINED | |
| 0xE00FF000-0xE00FFFFF | ROM table | |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{RRLF}$ The addresses of the optional debug resources are:

| Address range | Debug Resource | |
|---|---|---|
| 0xE0000000-0xE0000FFF | *Instrumentation Trace Macrocell* (ITM) | |
| 0xE0001000-0xE0001FFF | *Data Watchpoint and Trace* (DWT) Unit | |
| 0xE0002000-0xE0002FFF | *Flashpatch and Breakpoint* Unit (FPB) | |
| 0xE0003000-0xE0003FFF | *Performace Monitor Unit* (PMU) | |
| 0xE000E000-0xE000EFFF | Secure SCS | |
| | 0xE000EC00-0xE000ED8F | *Secure System Control Block* (SCB) |
| | 0xE000EDF0-0xE000EEFF | *Secure Debug Control Block* (DCB) |
| 0xE002E000-0xE002EFFF | Non-secure SCS | |
| | 0xE002EC00-0xE002ED8F | *Non-secure System Control Block* (SCB) |
| | 0xE002EDF0-0xE002EEFF | *Non-secure Debug Control Block* (DCB) |
| 0xE0040000-0xE0040FFF | *Trace Port Interface Unit* (TPIU), when not implemented as a shared resource otherwise reserved. | |
| 0xE0041000-0xE0041FFF | *Embedded Trace Macrocell* (ETM) | |
| 0xE0042000-0xE00FEFFF- | IMPLEMENTATION DEFINED | |
| 0xE00FF000-0xE00FFFFF | ROM table | |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**.*

See also:

B13.1 *Instrumentation Trace Macrocell* on page 325.

B13.2 *Data Watchpoint and Trace unit* on page 334.

B13.5 *Flash Patch and Breakpoint unit* on page 359.

Chapter B7 *The System Address Map* on page 240.

B12.2.2 *Debug System registers* on page 283.

B13.4 *Trace Port Interface Unit* on page 357.

B13.3 *Embedded Trace Macrocell* on page 356.

B12.2.1 *ROM table* on page 281.

B12.2 *Accessing debug features* on page 281.

### B12.1.3 Trace

$R_{LJVL}$ Trace can be generated by using the:

- *Embedded Trace Macrocell* (ETM).
- *Instrumentation Trace Macrocell* (ITM).
- *Data Watchpoint and Trace* (DWT) unit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM** || **ITM** || **DWT-T**. Note, ITM requires M.*

$R_{CZKQ}$ Trace can be generated by using the:

- *Embedded Trace Macrocell* (ETM).
- *Instrumentation Trace Macrocell* (ITM).
- *Data Watchpoint and Trace* (DWT) unit.
- *Performance Monitoring Unit* (PMU).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **ETM** || **ITM** || **DWT-T** || **PMU**. Note, ITM requires M.*

$R_{NFVB}$ A debug implementation that generates trace includes a trace sink, such as a TPIU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - (**ETM** || **ITM** || **DWT-T**) && **TPIU**. Note, ITM requires M.*

$R_{TBHB}$ A debug implementation that includes the PMU and generates trace includes a trace sink, such as a TPIU.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - (**ETM** || **ITM** || **DWT-T** || **PMU**) && **TPIU**. Note, ITM requires M.*

$I_{RJKJ}$ A TPIU can be either the Armv8-M TPIU implementation, or an external system resource.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM** || **ITM** || **DWT-T**.*

See also:

ITM and DWT Packet Protocol Specification.

The applicable ETM Architecture Specification.

## B12.2 Accessing debug features

$R_{WVSZ}$ The mechanism by which an external debugger accesses the PE and system is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{QPHR}$ A debugger can use a *Debug Access Port* (DAP) interface, such as that provided by the *Arm®Debug Interface v5 Architecture Specification*(ADIv5), to interrogate a system for memory access ports (MEM-APs). The base register in a memory access port provides the address of the ROM table, or the first of a series of ROM tables in a ROM table hierarchy. The memory access port can then fetch the ROM table entries. Arm recommends implementation of an ADIv5 DAP for compatibility with tools.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{WPGQ}$ Writes from a DAP are complete when the DAP reports them as complete.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{WCQK}$ For SCS registers, a write from a DAP is complete when the write has completed and the SCS register has been updated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{JRHS}$ Software configures and controls the debug model through memory-mapped registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

See also:

B12.2.1 *ROM table* .

B12.3.5 *DAP access permissions* on page 297.

The *Arm®Debug Interface v5 Architecture Specification*.

### B12.2.1 ROM table

$I_{XFVN}$ The ROM table is a table of entries providing a mechanism to identify the debug infrastructure that is supported by the implementation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{FWPG}$ The ROM table indicates the implemented debug components, and the position of those components in the memory map. See the *Arm®Debug Interface v5 Architecture Specification* for the format of a ROM table entry.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{PHJJ}$ For an Armv8-M ROM table, all entry offsets are negative. The ROM table entry points to the top of a 4KB page, the offset points to the bottom of that page that contains the Peripheral and Component ID registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{GPPX}$ The ROM table is implemented if any other debug component is implemented or a Debug Access Port is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{BQSP}$ Bit[0] of the ROM table entries indicates whether the corresponding debug component is implemented and is accessible through the PPB at the indicated address. If the corresponding debug component is not implemented, this bit has a value of 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{NDQW}$ If a debug component is implemented, debug registers can provide additional information about the implemented features of that debug component.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{DPVG}$ The format of the ROM table is:

| 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 unit at 0xE0001000 |
| 0x008 | 0xFFF03002 or 0xFFF03003 | ROMFPB | Points to the Flash Patch and Breakpoint unit at 0xE0002000 |
| 0x00C | 0xFFF01002 or 0xFFF01003 | ROMITM | Points to Instrumentation Trace unit at 0xE0000000. |
| 0x010 | 0xFFF41002 or 0xFFF41003 | ROMTPIU | Points to the Trace Port Interface Unit. |
| 0x014 | 0xFFF42002 or 0xFFF42003 | ROMETM | Points to the Embedded Trace Macrocell. |
| - | 0x00000000 | End | End of table marker. It is IMPDEF whether the table is extended with pointers to other system debug resources. The table entries always terminate with a null entry. |
| 0x020 - 0xEFC | - | Not used | Reserved for additional ROM table entries. |
| 0xF00 - 0xFC8 | - | Reserved | Reserved, not used for ROM table entries. |
| 0xFCC | 0x00000001 | MEMTYPE | Bit [0] is set to 1 to indicate that resources other than those listed in the ROM table are accessible in the same 32-bit address space, using the DAP.Bits [31:1] of the MEMTYPE entry are RES0. |
| 0xFD0 | IMP DEF | PIDR4 | CIDRx values are fully defined for the ROM table, and are CorseSight compliant. PIDRx values are CoreSight compliant or RAZ. |
| 0xFD4 | 0 | PIDR5 | |
| 0xFD8 | 0 | PIDR6 | |
| 0xFDC | 0 | PIDR7 | |
| 0xFE0 | IMP DEF | PIDR0 | |
| 0xFE4 | IMP DEF | PIDR1 | |
| 0xFE8 | IMP DEF | PIDR2 | |
| 0xFEC | IMP DEF | PIDR3 | |
| 0xFF0 | 0x0000000D | CIDR0 | |
| 0xFF4 | 0x00000010 | CIDR1 | |
| 0xFF8 | 0x00000005 | CIDR2 | |
| 0xFFC | 0x000000B1 | CIDR3 | |

Accesses to the ROMITM cannot cause a non-existent memory exception.

It is IMPLEMENTATION DEFINED whether the ETM and TPIU are a shared resource and whether the resource is managed by the local PE or a different resource.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB and those indicated in the table**.*

$R_{ZZGJ}$ The format of the ROM table is:

| 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 unit at 0xE0001000 |
| 0x008 | 0xFFF03002 or 0xFFF03003 | ROMFPB | Points to the Flash Patch and Breakpoint unit at 0xE0002000 |
| 0x00C | 0xFFF01002 or 0xFFF01003 | ROMITM | Points to Instrumentation Trace unit at 0xE0000000. |
| 0x010 | 0xFFF41002 or 0xFFF41003 | ROMTPIU | Points to the Trace Port Interface Unit. |
| 0x014 | 0xFFF42002 or 0xFFF42003 | ROMETM | Points to the Embedded Trace Macrocell. |
| 0x018 | 0xFFF42002 or 0xFFF42003 | ROMPMU | Points to the Performance Monitoring Unit. |
| - | 0x00000000 | End | End of table marker. It is IMPDEF whether the table is extended with pointers to other system debug resources.<br>The table entries always terminate with a null entry. |
| 0x020 - 0xEFC | - | Not used | Reserved for additional ROM table entries. |
| 0xF00 - 0xFC8 | - | Reserved | Reserved, not used for ROM table entries. |
| 0xFCC | 0x00000001 | MEMTYPE | Bit [0] is set to 1 to indicate that resources other than those listed in the ROM table are accessible in the same 32-bit address space, using the DAP.Bits [31:1] of the MEMTYPE entry are RES0. |
| 0xFD0 | IMP DEF | PIDR4 | CIDRx values are fully defined for the ROM table, and are CorseSight compliant. |
| 0xFD4 | 0 | PIDR5 | PIDRx values are CoreSight compliant or RAZ. |
| 0xFD8 | 0 | PIDR6 | |
| 0xFDC | 0 | PIDR7 | |
| 0xFE0 | IMP DEF | PIDR0 | |
| 0xFE4 | IMP DEF | PIDR1 | |
| 0xFE8 | IMP DEF | PIDR2 | |
| 0xFEC | IMP DEF | PIDR3 | |
| 0xFF0 | 0x0000000D | CIDR0 | |
| 0xFF4 | 0x00000010 | CIDR1 | |
| 0xFF8 | 0x00000005 | CIDR2 | |
| 0xFFC | 0x000000B1 | CIDR3 | |

Accesses to the ROMITM cannot cause a non-existent memory exception.

It is IMPLEMENTATION DEFINED whether the ETM and TPIU are a shared resource and whether the resource is managed by the local PE or a different resource.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB and those indicated in the table**.*

$R_{RGVM}$ The entry 0x00000000 is the end-of-table marker.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{JDGV}$ If a PMU is implemented, the end-of-table marker is 0x1C and has a value of 0x00000000. It is IMPLEMENTATION DEFINED whether the table is extended with pointers to other System debug resources.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB && PMU**.*

See also:

B12.2.3 *CoreSight and identification registers* on page 284.

## B12.2.2 Debug System registers

$R_{RHDW}$ The debug provision in the *System Control Block* (SCB) comprises:

- Two handler-related flag bits, ICSR.ISRPREEMPT and ICSR.ISRPENDING.
- The DFSR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

See also:

Chapter D1, Register Specification.

Debug Control Block.

### B12.2.3 CoreSight and identification registers

$I_{CMLH}$ Arm recommends that CoreSight-compliant ID registers are implemented to allow identification and discovery of the components to a debugger.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{CBCM}$ The address spaces that are reserved in each of the debug components for IMPLEMENTATION DEFINED ID registers and CoreSight compliance are:

| Debug Component | Space reserved for ID registers | Space reserved for CoreSight compliance |
|---|---|---|
| ITM | 0xE0000FD0-0xE0000FFC | 0xE0000FA0-0xE0000FCC |
| DWT | 0xE0001FD0-0xE0001FFC | 0xE0001FA0-0xE0001FCC |
| FPB | 0xE0002FD0-0xE0002FFC | 0xE0002FA0-0xE0002FCC |
| SCS | 0xE000EFD0-0xE000EFFC | 0xE000EFA0-0xE000EFCC |
| TPIU | 0xE0040FD0-0xE0040FFC | 0xE0040FA0-0xE0040FCC |
| ETM | 0xE0041FD0-0xE0041FFC | 0xE0041FA0-0xE0041FCC |
| ROM table | 0xE00FFFD0-0xE00FFFFC | 0xE00FFFA0-0xE00FFFCC |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{QCWD}$ The address spaces that are reserved in each of the debug components for IMPLEMENTATION DEFINED ID registers and CoreSight compliance are:

| Debug Component | Space reserved for ID registers | Space reserved for CoreSight compliance |
|---|---|---|
| ITM | 0xE0000FD0-0xE0000FFC | 0xE0000FA0-0xE0000FCC |
| DWT | 0xE0001FD0-0xE0001FFC | 0xE0001FA0-0xE0001FCC |
| FPB | 0xE0002FD0-0xE0002FFC | 0xE0002FA0-0xE0002FCC |
| PMU | 0xE0003FD0-0xE0003FFC | 0xE0003FA0-0xE0003FFC |
| SCS | 0xE000EFD0-0xE000EFFC | 0xE000EFA0-0xE000EFCC |
| TPIU | 0xE0040FD0-0xE0040FFC | 0xE0040FA0-0xE0040FCC |
| ETM | 0xE0041FD0-0xE0041FFC | 0xE0041FA0-0xE0041FCC |
| ROM table | 0xE00FFFD0-0xE00FFFFC | 0xE00FFFA0-0xE00FFFCC |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**.*

$R_{VWSX}$ For the ROM table, the ID register space is used for a set of CoreSight-compliant ID registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{HXDK}$ For all components other than the ROM table, if the registers in the ID register space are not used for ID registers they are RAZ.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{VQPM}$ If CoreSight-compliant ID registers are implemented, the Class field in Component ID Register 1 is:

- `0x1` for the ROM table.
- `0x9` for other components.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{HQSR}$ The Part number in the PIDR registers must be assigned a unique value for each implementation, or Unique Component Identifier, as with all other CoreSight components.

CoreSight permits that two or more functionally different components are permitted to share the same Part number, so long as they have different values of the DDEVTYPE or DDEVARCH registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{CTBF}$ The Part number in the PIDR registers do not need to be unique for different implementation options of the same part.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

## B12.3 Debug authentication interface

$I_{GWTN}$ The following pseudocode functions provide an abstracted description of the authentication interface:

- `ExternalInvasiveDebugEnabled()`.
- `ExternalSecureInvasiveDebugEnabled()`.
- `ExternalNoninvasiveDebugEnabled()`.
- `ExternalSecureNoninvasiveDebugEnabled()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{SWWT}$ For an implementation using the CoreSight signals **DBGEN**, **NIDEN**, **SPIDEN**, and **SPNIDEN**:

- `ExternalInvasiveDebugEnabled()` returns TRUE if **DBGEN** is asserted.
- `ExternalSecureInvasiveDebugEnabled()` returns TRUE if both **DBGEN** and **SPIDEN** are asserted.
- `ExternalNoninvasiveDebugEnabled()` returns TRUE if either **NIDEN** or **DBGEN** is asserted.
- `ExternalSecureNoninvasiveDebugEnabled()` returns TRUE if both of the following conditions apply:
  - Either **NIDEN** or **DBGEN** is asserted.
  - Either **SPNIDEN** or **SPIDEN** is asserted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{FCPK}$ The unprivileged debug capabilities enabled by DAUTHCTRL.UIDEN being set to 1 are available regardless of the state of the following:

- DAUTHCTRL.INTSPNIDEN.
- DAUTHCTRL.SPNIDENSEL.
- DAUTHCTRL.INTSPIDEN.
- DAUTHCTRL.SPIDENSEL.
- **DBGEN**.
- **SPIDEN**.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, S is required for Secure Behavior.*

$R_{HVGN}$ For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is FALSE, then `ExternalSecureInvasiveDebugEnabled()` is FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{JWCS}$ For any implementation of the authentication interface, if `ExternalNoninvasiveDebugEnabled()` is FALSE, then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{XCMD}$ For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is TRUE, then `ExternalNoninvasiveDebugEnabled()` is TRUE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{LCHH}$ For any implementation of the authentication interface, if `ExternalSecureInvasiveDebugEnabled()` is TRUE, then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{MSRG}$ Secure self-hosted debug is controlled by the authentication interface. The pseudocode function `ExternalSecureSelfHostedDebugEnabled()` provides an abstracted description of this authentication interface.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{GLWM}$ Between a change to the debug authentication interface and a following Context synchronization event, it is UNPREDICTABLE whether the PE uses the old or the new values.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

See also:

B12.3.1 *Halting debug authentication* .

B12.3.3 *DebugMonitor exception authentication* on page 294.

B12.3.2 *Non-invasive debug authentication* on page 292.

B12.3.5 *DAP access permissions* on page 297.

### B12.3.1 Halting debug authentication

$I_{DMFG}$ Halting debug authentication is controlled by the IMPLEMENTATION DEFINED authentication interface function `ExternalInvasiveDebugEnabled()`, and if the Security Extension is implemented, the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureInvasiveDebugEnabled()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, External Secure invasive debug requires S.*

$R_{JJJK}$ Unless otherwise stated Halting is prohibited in all states if the function `ExternalInvasiveDebugEnabled()` returns FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{PHWV}$ Unless otherwise stated Halting is prohibited in all states if the function `ExternalInvasiveDebugEnabled()` returns FALSE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && !UDE**.*

$R_{JXTX}$ When the PE is halted, the PE behaves as if `ExternalInvasiveDebugEnabled()` is TRUE. The pseudocode function `HaltingDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{LMHR}$ When the PE is halted, the PE behaves as if `ExternalInvasiveDebugEnabled()` is TRUE. The pseudocode function `HaltingDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && !UDE**.*

$I_{BCZM}$ If the Security Extension is not implemented, there are two Halting debug authentication modes:

| `ExternalInvasiveDebugEnabled()` | DHCSR.S_HALT | Halting debug authentication mode |
|---|---|---|
| FALSE | 0 | Halting is prohibited. |
| FALSE | 1 | Halting is allowed. |
| TRUE | X | Halting is allowed. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug***

***&& !S***.

$R_{BMRJ}$ Halting is prohibited in Secure state if any of:

- `ExternalInvasiveDebugEnabled()` returns FALSE.
- DAUTHCTRL.SPIDENSEL is set to 1 and DAUTHCTRL_S.INTSPIDEN is set to 0.
- DAUTHCTRL.SPIDENSEL is set to 0 and `ExternalSecureInvasiveDebugEnabled()` returns FALSE.

The pseudocode function `SecureHaltingDebugAllowed` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{HXQH}$ Halting is prohibited in Secure state if any of:

- `ExternalInvasiveDebugEnabled()` returns FALSE.
- DAUTHCTRL.SPIDENSEL is set to 1 and DAUTHCTRL.INTSPIDEN is set to 0.
- DAUTHCTRL.SPIDENSEL is set to 0 and `ExternalSecureInvasiveDebugEnabled()` returns FALSE.

The pseudocode function `SecureHaltingDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && S && !UDE**.*

$R_{QTBK}$ Halting is prohibited in unprivileged modes in Secure state if all of:

- `SecureHaltingDebugAllowed()` returns FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{BDNX}$ Halting in unprivileged modes is prohibited in Secure state if all of:

- `SecureHaltingDebugAllowed()` returns FALSE.
- DAUTHCTRL_S.UIDEN is set to 0.

The pseudocode function `UnprivHaltingDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE && Halting debug && S**.*

$R_{KBKM}$ If the PE is in non-Debug state the following condition is true:

- DHCSR.S_SDE reads as one if any one of the following of true, and reads as zero otherwise:
  - `SecureHaltingDebugAllowed()` returns TRUE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, S is required for Secure Behavior.*

$R_{CVKR}$ If UDE is implemented and the PE is in non-Debug state the following conditions are true:

- DHCSR.S_SDE reads as one if any one of the following of true, and reads as zero otherwise:
  - `SecureHaltingDebugAllowed()` returns TRUE.
  - `UnprivHaltingDebugAllowed(TRUE)` returned TRUE.
- DHCSR.S_SUIDE reads as one if both of the following are true and reads as zero otherwise:
  - `UnprivHaltingDebugAllowed(TRUE)` returned TRUE.
  - `SecureHaltingDebugAllowed()` returned FALSE.
- DHCSR.S_NSUIDE reads as one if both of the following are true and reads as zero otherwise:
  - `UnprivHaltingDebugAllowed(FALSE)` returned TRUE.
  - `HaltingDebugAllowed()` returned FALSE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, S is required for Secure behavior.*

$R_{KMXG}$ If the PE is in Debug state:

- DHCSR.S_SDE reads as one if either of the the following is true, and reads as zero otherwise:
  - The PE entered Debug state from Secure state.
  - The PE entered Debug state from Non-secure state when `SecureHaltingDebugAllowed()` returned TRUE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, S is required for Secure behavior.*

$R_{TNMM}$ If UDE is implemented and the PE is in Debug state:

- DHCSR.S_SDE reads as one if any one of the following is true, and reads as zero otherwise:
  - The PE entered Debug state from Secure state.
  - The PE entered Debug state from Non-secure state when `SecureHaltingDebugAllowed()` returned TRUE.
  - The PE entered Debug state from Non-secure state when `UnprivHaltingDebugAllowed(TRUE)` returned TRUE.
- DHCSR.S_SUIDE reads as one if the PE entered Debug state if both of the following are true and reads as zero otherwise:
  - `UnprivHaltingDebugAllowed(TRUE)` returns TRUE.
  - `SecureHaltingDebugAllowed()` returns FALSE.
- DHCSR.S_NSUIDE reads as one if the PE entered Debug state if both the following are true, and reads as zero otherwise:
  - `UnprivHaltingDebugAllowed(FALSE)` returns TRUE.
  - `HaltingDebugAllowed()` returned FALSE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, S is required for Secure behavior.*

$I_{VQJZ}$ The value of DHCSR.S_SDE, DHCSR.S_SUIDE and DHCSR.S_NSUIDE are updated when the PE is not halted as described by the function `UpdateDebugEnable()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug**. Note, S is required for Secure behavior. UDE is required for Unprivileged Debug.*

$R_{VHKZ}$ DAUTHCTRL_NS.UIDEN controls whether debug requests can be raised from unprivileged code in Non-secure state. DAUTHCTRL_S.UIDEN controls whether debug requests can be raised from unprivileged code in Secure state. When DAUTHCTRL_S.UIDEN is set to 1, DHCSR_NS.S_NSUIDEN has an Effective value of 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, S is required for Secure Behavior.*

$I_{LDTR}$ If the Security Extension is implemented, there are three Halting debug authentication modes:

| `HaltingDebugAllowed()` | DHCSR.S_SDE | **Halting debug authentication mode** |
|---|---|---|
| FALSE | X | Halting is prohibited. |
| TRUE | 0 | Halting is allowed in Non-secure state.<br>Halting is prohibited in Secure state. |
| | 1 | Halting is allowed. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$I_{FGCF}$ The following table describes whether Halting debug is permitted for Secure and Non-Secure state and privilege level with the effective value of the external signals.

| **DEBUGEN** | **SPIDEN** | **DAUTHCTRL _NS.UIDEN** | **DAUTHCTRL _S.UIDEN** | **Non-secure Privileged** | **Non-secure Unprivileged** | **Secure Privileged** | **Secure Unprivileged** |
|---|---|---|---|---|---|---|---|
| FALSE | X | 0 | 0 | Prohibited | Prohibited | Prohibited | Prohibited |
| FALSE | X | 1 | 0 | Prohibited | Permitted | Prohibited | Prohibited |
| FALSE | X | X | 1 | Prohibited | Permitted | Prohibited | Permitted |
| TRUE | FALSE | X | 0 | Permitted | Permitted | Prohibited | Prohibited |
| TRUE | FALSE | X | 1 | Permitted | Permitted | Prohibited | Permitted |
| TRUE | TRUE | X | X | Permitted | Permitted | Permitted | Permitted |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, S is required for Secure behavior.*

$R_{FXCB}$ When DHCSR.C_DEBUGEN == 0 or the PE is in a state in which Halting is prohibited, the PE does not enter Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, S is required for Secure behavior.*

See also:

`CanHaltOnEvent()`.

### B12.3.2 Non-invasive debug authentication

$R_{GFTG}$ Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function:

`ExternalNoninvasiveDebugEnabled()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{HXQD}$ Secure Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function:

`ExternalSecureNoninvasiveDebugEnabled()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{CFNB}$ When `HaltingDebugAllowed()` is TRUE:

- The PE behaves as if `ExternalNoninvasiveDebugEnabled()` returns TRUE.
- The pseudocode function `NoninvasiveDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{VRJJ}$ When `HaltingDebugAllowed()` is TRUE:

- the PE behaves as if `ExternalNoninvasiveDebugEnabled()` returns TRUE.
- The pseudocode function `NoninvasiveDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && !**UDE**.*

$R_{LNCF}$ When `HaltingDebugAllowed()` is TRUE or `UnprivHaltingDebugAllowed()` is TRUE, the PE behaves as if `ExternalNoninvasiveDebugEnabled()` returns TRUE. The pseudocode function `NoninvasiveDebugAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && **UDE**.*

$R_{QMRF}$ Non-invasive debug is prohibited if the functions `NoninvasiveDebugAllowed()` and `SecureNoninvasiveDebug` both return FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{PHPR}$ If the Security Extension is not implemented, there are two non-invasive debug authentication modes:

| `ExternalNon-invasiveDebugEnabled()` | `HaltingDebugAllowed()` | **Non-invasive debug authentication mode** |
|---|---|---|
| FALSE | FALSE | Non-invasive debug prohibited. |
| | TRUE | Non-invasive debug allowed. |
| TRUE | X | Non-invasive debug allowed. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{THCW}$ If the Unprivileged Debug Extension is implemented non-invasive debug has the following debug authentication modes:

| `ExternalNoninvasive-DebugEnabled()` | `HaltingDebugAllowed()` | `UnprivHalting-DebugAllowed()` | **Non-invasive debug authentication mode** |
|---|---|---|---|
| FALSE | FALSE | FALSE | Privileged and Unprivileged Non-invasive debug prohibited. |
| | TRUE | X | Privileged and Unprivileged Non-invasive debug allowed. |
| | FALSE | TRUE | Unprivileged Non-invasive debug allowed.<br>Privileged Non-invasive debug prohibited. |
| TRUE | X | X | Privileged and Unprivileged Non-invasive debug allowed. |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && **UDE**. Note, This applies to Non-secure state only.*

$I_{MLPS}$ Non-invasive debug of Secure operations is prohibited if any of the following are true:

- `NoninvasiveDebugAllowed()` returns FALSE.
- DHCSR.S_SDE is set to 0, DAUTHCTRL.SPNIDENSEL is set to 1 and DAUTHCTRL.INTSPIDEN is set to 0.
- DHCSR.S_SDE is set to 0, DAUTHCTRL.SPNIDENSEL is set to 0 and `ExternalNoninvasiveDebugEnabled` returns FALSE.

The pseudocode function `SecureNoninvasiveDebugAllowed()` shows this, if this function returns true Secure Non-invasive debug is permitted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{PNRC}$ If the Security Extension is implemented, there are three non-invasive debug authentication modes:

| Noninvasive-DebugEnabled() | SecureNon-invasiveDebugAllowed() | **Non-invasive debug authentication mode** |
|---|---|---|
| FALSE | X | Non-invasive debug prohibited. |
| TRUE | FALSE | Non-invasive debug of only Non-secure operations allowed.<br>Non-invasive debug of Secure operations prohibited. |
| | TRUE | Non-invasive debug allowed. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$I_{TLMK}$ If the Security Extension is implemented, there are three non-invasive debug authentication modes:

| Noninvasive-DebugEnabled() | SecureNon-invasiveDebugAllowed() | UnprivHalting-DebugAllowed(NS) | UnprivHalting-DebugAllowed(S) | **Non-invasive debug authentication mode** |
|---|---|---|---|---|
| FALSE | FALSE | FALSE | FALSE | All non-invasive debug prohibited. |
| | | TRUE | FALSE | Unprivileged Non-invasive Debug of Non-secure operations allowed. |
| | | X | TRUE | Unprivileged Non-invasive Debug of Secure and Non-secure operations allowed. |
| TRUE | FALSE | X | FALSE | Non-invasive debug of all non-secure operations allowed. |
| | | X | TRUE | Non-invasive debug of all non-secure, and unprivileged secure operations allowed. |
| | TRUE | X | X | All non-invasive debug allowed. |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && **S** && **UDE**.*

$R_{LXRK}$ The PE does not generate any trace or profiling data when non-invasive debug is prohibited.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{VYGT}$ If non-invasive debug of Secure operations is prohibited, the PE does not generate any trace or profiling data that contains secure information or is attributable to secure operations.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$R_{TWDH}$ If non-invasive debug is prohibited in the current Security state, an ETM behaves as described in the relevant ETM architecture.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S** && **ETM**.*

See also:

`NoninvasiveDebugAllowed()`.

`SecureNoninvasiveDebugAllowed()`.

B13.2.2 *DWT unit operation* on page 335.

### B12.3.3 DebugMonitor exception authentication

$R_{MXTM}$ DebugMonitor exception authentication is only available if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{LQCN}$ DebugMonitor exception authentication is controlled by the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureSelfHostedDebugEnabled()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{QJQN}$ Unless otherwise stated DebugMonitor exceptions are never generated for Secure operations if any of:

- DAUTHCTRL.SPIDENSEL is set to 1 and DAUTHCTRL.INTSPIDEN is set to 0.

- DAUTHCTRL.SPIDENSEL is set to 0 and `ExternalSecureSelfHostedDebugEnabled()` returns FALSE.

The pseudocode function `SecureDebugMonitorAllowed()` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{PWJZ}$ If UDE is implemented DAUTHCTRL.FSDMA allows the Secure DebugMonitor exception to be enabled independently of Halting debug and `ExternalSecureSelfHostedDebugEnabled()`. This field does not control the DebugMonitor exception permissions directly, instead DAUTHCTRL.FSDMA is used as an input into DEMCR.SDME, as described by `UpdateDebugEnable()` and `SecureDebugMonitorAllowed()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M** && **S** && **UDE**.*

$R_{CPPN}$ When a DebugMonitor exception is pending or active:

- DEMCR.SDME is set to 1 if `SecureDebugMonitorAllowed()` returned TRUE when a DebugMonitor exception became pending or active.
- DEMCR.SDME is zero otherwise.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{WXMG}$ When a DebugMonitor exception is not pending and is not active:

- DEMCR.SDME is set to 1 if `SecureDebugMonitorAllowed()` is TRUE.
- DEMCR.SDME is zero otherwise.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{RZXJ}$ If the Security Extension is implemented, there are two DebugMonitor exception authentication modes, which are controlled by DEMCR.SDME:

| DEMCR.SDME | Target State for DebugMonitor exception | DebugMonitor exception authentication mode |
|---|---|---|
| 0 | Non-secure | Non-secure DebugMonitor exception. |
| 1 | Secure | Secure DebugMonitor exception. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{YFPK}$ If DEMCR.SDME == 1, SHPR3.PRI_12 behaves as RAZ/WI when accessed from Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** && **S**.*

$R_{HXLX}$ When set to 1, DEMCR.MON_PEND remains set to 1 until either the DebugMonitor exception is taken or a write sets the field to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

See also:

`CanPendMonitorOnEvent()`.

### B12.3.4 Unprivileged DebugMonitor Authentication

$R_{CTNC}$ Setting DEMCR.UMON_EN to 1 permits a DebugMonitor Exception to be raised from the unprivileged mode where this would otherwise be prohibited by DEMCR.MON_EN being cleared to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE** && **M**.*

$R_{XLGC}$ When DEMCR.SDME and DEMCR.UMON_EN are set to 1, a DebugMonitor exception can be raised from unprivileged execution regardless of the current Security state of the PE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE** && **M**. Note, S is required for Secure behavior.*

$R_{SKDL}$ The following table shows whether the DebugMonitor exception can be raised for each Security state and privilege level.

| DEMCR.MON_EN | DEMCR.SDME | DEMCR.UMON_EN | Non-secure Privileged | Non-secure Unprivileged | Secure Privileged | Secure Unprivileged |
|---|---|---|---|---|---|---|
| 0 | x | 0 | No | No | No | No |
| 0 | 0 | 1 | No | Yes | No | No |
| 0 | 1 | 1 | No | Yes | No | Yes |
| 1 | 0 | x | Yes | Yes | No | No |
| 1 | 1 | x | Yes | Yes | Yes | Yes |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE** && **M**. Note, S is required for Secure behavior.*

See also

*DEMCR, Debug Exception Monitor Control Register*.

`ExternalSecureSelfHostedDebugEnabled()`.

`SecureDebugMonitorAllowed()`.

`CanPendMonitorOnEvent()`.

`UpdateDebugEnable()`.

### B12.3.5 DAP access permissions

$R_{BFSB}$ When `HaltingDebugAllowed()` returns TRUE the external debugger can access the whole physical address space.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{BQQQ}$ When `HaltingDebugAllowed()` returns TRUE or DAUTHCTRL.UIDAPEN (either bank) is set the external debugger can access the whole physical address space.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**. Note, DAUTHCTRL.UIDAPEN requires UDE.*

$R_{DVSN}$ Unless otherwise stated if `HaltingDebugAllowed()` = FALSE the DAP access permissions are:

| Address Range | Region or registers | `NoninvasiveDebugAllowed()` | | |
|---|---|---|---|---|
| | | | FALSE | TRUE |
| `0x00000000-0xDFFFFFFF` | Rest of Memory | | No access | No access |
| `0xE0000000-0xE00FFFFF` | PPB | | | |
| | `0xE00xxFB0-0xE00xxFB7` | CoreSight Software Lock registers | No access | RW |
| | `0xE00xxFD0-0xE00xxFFF` | All ID registers | RO | RO |
| | `0xE0000000-0xE0000FCF` | ITM | No access | RW |
| | `0xE0001000-0xE0001FCF` | DWT | No access | RW |
| | `0xE0040000-0xE0040FFF` | TPIU | RW | RW |
| | `0xE0041000-0xE0041FFF` | ETM | RW | RW |
| | `0xE0042000-0xE00FEFFF` | IMPDEF | IMPDEF | IMPDEF |
| | `0xE00FF000-0xE00FFFFF` | ROM table | RO | RO |
| | All other PPB regions and registers | | No access | No access |
| `0xE0100000-0xFFFFFFFF` | Vendor_SYS | | No access | RW |

`0xE00xxFB0-0xE00xxFB7` is for each debug component implementing the CoreSight Software Lock registers. These registers are optional. `0xE00xxFD0-0xE00xxFFF` for each debug component implementing the CoreSight ID registers. These registers are optional.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{TSKQ}$ Unless otherwise stated if `HaltingDebugAllowed()` = FALSE the DAP access permissions are:

| Address Range | Region or registers | `NoninvasiveDebugAllowed()` | | |
|---|---|---|---|---|
| | | | **FALSE** | **TRUE** |
| `0x00000000-0xDFFFFFFF` | Rest of Memory | | No access | No access |
| `0xE0000000-0xE00FFFFF` | PPB | | | |
| | `0xE00xxFB0-0xE00xxFB7` | CoreSight Software Lock registers | No access | RW |
| | `0xE00xxFD0-0xE00xxFFF` | All ID registers | RO | RO |
| | `0xE0000000-0xE0000FCF` | ITM | No access | RW |
| | `0xE0001000-0xE0001FCF` | DWT | No access | RW |
| | `0xE0003000-0xE0003FFF` | PMU | No access | RW |
| | `0xE0040000-0xE0040FFF` | TPIU | RW | RW |
| | `0xE0041000-0xE0041FFF` | ETM | RW | RW |
| | `0xE0042000-0xE00FEFFF` | IMPDEF | IMPDEF | IMPDEF |
| | `0xE00FF000-0xE00FFFFF` | ROM table | RO | RO |
| | All other PPB regions and registers | | No access | No access |
| `0xE0100000-0xFFFFFFFF` | Vendor_SYS | | No access | RW |

`0xE00xxFB0-0xE00xxFB7` is for each debug component implementing the CoreSight Software Lock registers. These registers are optional. `0xE00xxFD0-0xE00xxFFF` for each debug component implementing the CoreSight ID registers. These registers are optional.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**. Note, PMU requires the PMU extension, DAUTHCTRL.UIDAPEN requires UDE.*

$R_{FFPN}$ The DAP is capable of requesting Secure and Non-secure accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$R_{VDPK}$ The DAP is capable of requesting Privileged and Unprivileged accesses.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && **UDE**.*

$R_{KBQZ}$ A DAP memory access calls `MemD_with_priv_security()` and `DAPCheck()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB**. Note, Secure accesses require S and unprivileged accesses require UDE.*

$I_{PQSV}$ The architecture does not describe how a DAP requests Secure or Non-secure memory accesses. In the recommended ADIv5 Memory Access Port (MEM-AP), Arm recommends that:

- CSW[30], CSW.Prot[6], selects a Secure or Non-secure access:
  - **0**: Request a Secure access.
  - **1**: Request a Non-secure access.
- CSW[23], CSW.SPIDEN, is Read-As-One. This is because the DAP can always request a Secure access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$I_{DPHD}$ In a CoreSight DAP, the **SPIDEN** input to the Armv8-M MEM-AP is independent of the SPIDEN input of the PE, and must be tied HIGH.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

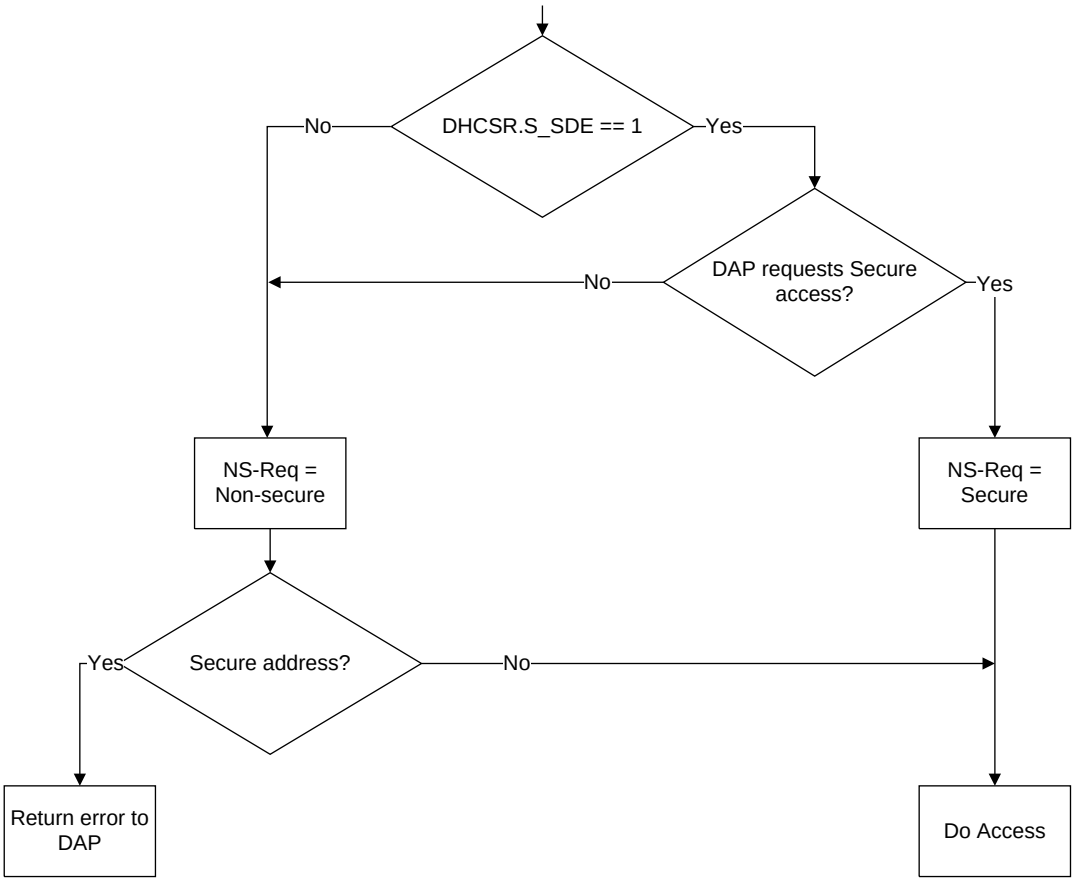
$R_{JHBC}$ If DHCSR.S_SDE == 1, and the DAP requests a Secure access, NS-Req is set to Secure.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$R_{LVBG}$ If either DHCSR.S_SDE == 0 or the DAP requests a Non-secure access, NS-Req set to Non-secure.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$R_{WMRR}$ DAP accesses are checked by the IDAU and the SAU, if applicable. That is, if NS-Req on a DAP access specifies Non-secure access, and the IDAU or SAU prohibits Non-secure access to the address, an error response is returned to the DAP.




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB && S**.*

$R_{VTTN}$ Unless otherwise stated DAP accesses are not checked by the MPU.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB && MPU**.*

$R_{FDCQ}$ DAP accesses to the SCS registers ignore NS-Req.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB && S**.*

$R_{SSVN}$ Permitted DAP accesses to Secure SCS registers in the range `0xE000E000-0xE000EFFF` are affected by the values of DHCSR.S_SDE, DSCSR.SBRSELEN, and DSCSR.SBRSEL, as well as by the current Security state of the PE. The following table shows the effect of these factors on the register being viewed.

| DHCSR.S_SDE | DSCSR.SBRSELEN | DSCSR.SBRSEL | **Current Security state of the PE** | **View of register accessed** |
|---|---|---|---|---|
| 0 | X | X | X | Non-secure. |
| 1 | 1 | 0 | X | Non-secure. |
| 1 | 1 | 1 | X | Secure. |
| 1 | 0 | X | Non-secure. | Non-secure. |
| 1 | 0 | X | Secure. | Secure. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB** && **S**.*

$R_{HXMG}$ Permitted DAP accesses to the region `0xE002E000-0xE002EFFF` are RAZ/WI if the access is privileged and return an error if the access is unprivileged.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{HBGQ}$ An Armv8.1-M PE with UDE extends the existing DAP access regime for the Main Extension.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DB** && **UDE**.*

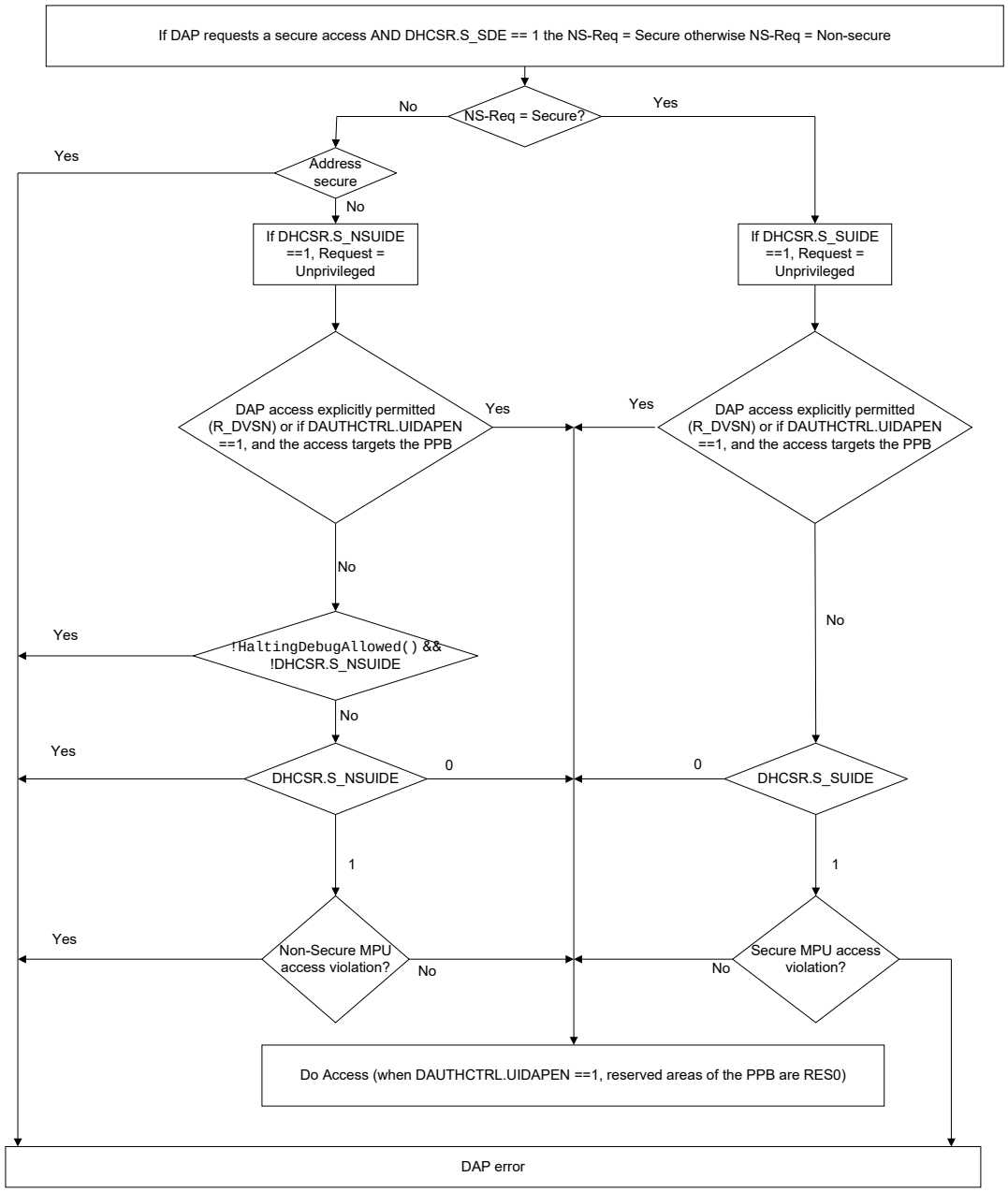
$R_{RXKV}$ When DAUTHCTRL.UIDAPEN is set, unprivileged debugger accesses to reserved locations within the PPB are treated as RES0 and do not return a DAP error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE**.*

$R_{MRPC}$ A privileged DAP request through the Unprivileged Debug Extension mechanism is demoted to an unprivileged access and is subject to MPU checks. Both privileged and unprivileged requests are subject to MPU checks.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE** && **MPU**.*

$R_{WSQR}$ The DAP access process is extended when the Unprivileged Debug Extension is implemented as shown in the following diagram.




*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **UDE** && **MPU**.*

See also:

B3.14 *Secure address protection* on page 95.

Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

# B12.4 Debug event behavior

## B12.4.1 About debug events

$I_{CBWT}$ An event that is triggered for debug reasons is known as *a debug event*.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{PQKW}$ A debug event that is not ignored causes one of the following to occur:

- If Halting debug is implemented and enabled, entry to Debug state.
- A HardFault exception.
- Lockup.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**. Note, entry to Debug state requires Halting Debug.*

$R_{QLTQ}$ A debug event that is not ignored, can cause a DebugMonitor exception to occur.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{MNKP}$ The HardFault exceptions or Lockup that are caused by debug events are generated by:

- A BKPT instruction that is executed when the PE can neither halt nor generate a DebugMonitor exception.
- In some circumstances, the FPB.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**. Note, an FPB requires FPB.*

$R_{WCPW}$ The debug events are as follows.

| Debug event | Actions |
|---|---|
| Step | Halt or DebugMonitor exception. |
| Halt Request | Halt |
| Breakpoint | Halt, DebugMonitor exception, or Hardfault. |
| Watchpoint | Halt or DebugMonitor exception. |
| Vector catch | Halt only |
| External | Halt or DebugMonitor exception. |

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, a DebugMonitor exception requires M. Halt requires Halting Debug.*

$R_{LNPG}$ The Armv8.1-M architecture adds the following debug event:

| Debug event | Actions |
|---|---|
| PMU Overflow | Halt |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{LDRZ}$ The DFSR contains status bits for each debug event. These bits are set to 1 when a debug event causes the PE to halt or generate a DebugMonitor exception, and are then write-one-to-clear.

The following table shows which bit is set for each debug event.

| Event cause | DFSR bit |
|---|---|
| Step | HALTED |
| Halt request | HALTED |
| Breakpoint | BKPT |
| Watchpoint | DWTTRAP |
| Vector catch | VCATCH |
| External | EXTERNAL |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** || **Halting debug**.*

$R_{HLXL}$ The DFSR contains status bits for each debug event. These bits are set to 1 when a debug event causes the PE to halt or generate a DebugMonitor exception, and are then write-one-to-clear.

The following table shows which bit is set for the PMU overflow event.

| Event cause | DFSR bit |
|---|---|
| PMU Overflow | PMU |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{HNRV}$ It is IMPLEMENTATION DEFINED whether the DFSR debug event bits are updated when an event is ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$I_{NSMV}$ Debug events are either synchronous or asynchronous.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{VSVN}$ The synchronous debug events are:

- Breakpoint debug events, caused by execution of a `BKPT` instruction or by a match in the FPB.
- Vector catch debug events, caused when one or more DEMCR.VC_* bits are set to 1, and the PE takes the corresponding exception.
- Step debug events, caused by DHCSR.C_STEP or DEMCR.MON_STEP.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{PVGM}$ A single instruction can generate several synchronous debug events.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{WJFB}$ Synchronous debug events are associated with the instruction that generated them and are taken instead of executing the instruction. The PE does not generate any other synchronous exception or debug event that might have occurred as a result of executing the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{RNRD}$ The Step debug event is taken on the instruction following the instruction being stepped. This means that prioritization of the event applies relative to any other exception or debug event for the following instruction, not for the instruction being stepped.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{JSPS}$ If multiple synchronous debug events and exceptions are generated on the same instruction, they are prioritized as follows:

1. Halt request (halting only), including where DHCSR.S_HALT is set by DHCSR.C_STEP of the previous instruction.
2. Highest-priority pending exception that is eligible to be taken. If the Main Extension is implemented, this might be a DebugMonitor exception, if DEMCR.MON_PEND == 1. This includes where

DEMCR.MON_PEND is set by DEMCR.MON_STEP of the previous instruction.

3. Vector catch.
4. Fault from an instruction fetch, including synchronous BusFault error.
5. Breakpoint that is signaled by an FPB unit.
6. BKPT instruction or other exception that results from decoding the instructions. This includes the cases where exceptions from the instruction are UNDEFINED, an unimplemented or disabled coprocessor is targeted, or the EPSR.T bit has a value of 1.
7. Other synchronous exception that is generated by executing the instruction, including an exception that is generated by a memory access that is generated by the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**. Note, not all of the debug features listed might be implemented in a particular implementation.*

$R_{BQVF}$ The highest-priority synchronous debug event is reported in the DFSR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{FWQQ}$ It is UNPREDICTABLE whether synchronous debug events that occur on the same instruction as a debug event with a higher priority are reported in the DFSR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{TKRS}$ The asynchronous debug events are:

- Watchpoint debug events caused by a match in the DWT, including instruction address match watchpoints.
- Halt request debug events, where either:
  - A debugger write that has set DHCSR.C_HALT to 1 and DHCSR.C_DEBUGEN set to 1.
  - A software write that sets DHCSR.C_HALT to 1 when DHCSR.C_DEBUGEN was set to 1.
- External debug request debug events caused by assertion of an IMPLEMENTATION DEFINED external debug request.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{BGRM}$ The Armv8.1-M architecture adds the following asynchronous debug event:

- PMU Overflow caused by an overflow of a PMU counter that is configured to generate an interrupt.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{MRMC}$ When DHCSR.C_DEBUGEN == 0 or the PE is in a state in which halting is prohibited, DHCSR.C_HALT and DHCSR.C_STEP are ignored, and these bits have an Effective value of 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

See also:

B3.13 *Priority model* on page 91.

Halting debug.

DebugMonitor exception.

B12.4.3 *Vector catch* on page 312.

B14.6 *Interrupts and Debug events* on page 369.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

`GenerateDebugEventResponse()`.

## Halting debug

$R_{WLCF}$ Setting the DHCSR.C_DEBUGEN bit to 1 enables Halting debug.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{RZTG}$ A debug event sets DHCSR.C_HALT to 1 if all of the following conditions apply:

- The debug event supports generating entry to Debug state.
- DHCSR.C_DEBUGEN == 1.
- Unless otherwise stated, halting is allowed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{THLS}$ If DHCSR.C_HALT has a value of 1 and halting is allowed, the PE halts and enters Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{FKWB}$ A debug event that sets DHCSR.C_HALT to 1 pends entry to Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{MXLF}$ A debug event might set DHCSR.C_HALT and remain pending through execution in a mode or state where Halting debug is prohibited, which might not be a finite time. If halting is prohibited in Secure state and allowed in Non-secure state, then on transition from Secure to Non-secure state by an exception entry, exception return, Non-secure function call or function return, if DHCSR.C_HALT has a value of 1, the PE halts and enters Debug state before the first instruction executed in Non-secure state completes its execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{XSRJ}$ If DHCSR.C_HALT has a value of 1 or **EDBGRQ** is asserted before a Context synchronization event, and halting is allowed after the Context synchronization event, then the PE halts and enters Debug state before the first instruction following the Context synchronization event completes its execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug** || **EDBGRQ**.*

$R_{JXQF}$ DFSR is updated at the same time as the PE sets DHCSR.C_HALT to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{TXWB}$ If an instruction that is being stepped or an instruction that generates a debug event reads DFSR or DHCSR, the value that is read for the relevant DFSR bit or for DHCSR.C_HALT is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{FRJC}$ For asynchronous debug events, if halting is allowed, the PE enters Debug state in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{VJKX}$ Entering Debug state has no architecturally defined effect on the Event Register and exclusive monitors.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{JNGH}$ DHCSR.C_SNAPSTALL might allow imprecise entry into the Debug state, for example by forcing any stalled load or store instructions to be abandoned.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{BTBJ}$ If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, DHCSR.C_SNAPSTALL is ignored and has an Effective value of 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{HLNF}$ If DHCSR.S_SDE == 0, DHCSR.C_SNAPSTALL ignores writes from the debugger.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{CBLC}$ If UDE is implemented and DHCSR.S_SUIDE == 1, debugger writes to DHCSR.C_SNAPSTALL are ignored.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE && S**.*

$R_{RKBK}$ When the PE is in a state in which halting is prohibited, if DHCSR.C_HALT == 1 and DHCSR.C_DEBUGEN == 1, then DHCSR.C_HALT remains set unless it is cleared by a direct write to DHCSR. If the PE enters a state in which halting is allowed while DHCSR.C_HALT is set to 1, then the PE enters Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

See also:

*DHCSR, Debug Halting Control and Status Register*.

B12.4.2 *Debug stepping* on page 308.

B12.5 *Debug state* on page 318.

## DebugMonitor exception

$I_{DPCC}$ The DebugMonitor exception is only available if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{ZBSJ}$ The DebugMonitor exception is enabled when the DEMCR.MON_EN bit is set to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{HBZF}$ From Armv8.1-M the DebugMonitor exception can be enabled for unprivileged mode when DEMCR.UMON_EN is set to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M && UDE**.*

$R_{PPLF}$ A debug event sets DEMCR.MON_PEND to 1 if all of the following conditions apply:

- The debug event supports generating DebugMonitor exceptions and does not generate an entry to Debug state.
- DEMCR.MON_EN == 1.
- DEMCR.SDME == 1 for Secure state DebugMonitor exceptions.
- The DebugMonitor exception group priority is greater than the current execution priority.

The function `CanPendMonitorOnEvent()` describes this.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{VMDP}$ If UDE is implemented a debug event sets DEMCR.MON_PEND to 1 if all of the following conditions apply:

- The debug event supports generating DebugMonitor exceptions and does not generate an entry to Debug state.
- DEMCR.SDME == 1 for Secure state DebugMonitor exceptions.
- DEMCR.UMON_EN == 1 and `UnprivHaltingDebugAllowed()` returns TRUE, or DEMCR.MON_EN == 1 and the PE is in an unprivileged mode.
- The DebugMonitor exception group priority is greater than the current execution priority.

The function `CanPendMonitorOnEvent()` describes this.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M && UDE**.*

$R_{XNMW}$ If a Debug event does not generate an entry to Debug state and DEMCR.MON_EN is set to 0, or the DebugMonitor exception group priority value is lower than the current execution priority, or DEMCR.SDME == 0 and the DebugMonitor exception was generated in Secure state:

- The PE escalates a DebugMonitor synchronous exception that is generated by executing a `BKPT` instruction to a HardFault.
- The PE might set DEMCR.MON_PEND to 1 for a watchpoint debug event.
- The PE ignores the other debug events.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{VJJH}$ From Armv8.1-M, and when UDE is implemented, a Debug event that does not generate an entry to Debug state, and where DEMCR.MON_EN is set to 0, or DEMCR.UMON_EN is set to 0 and the current mode is privileged, or the DebugMonitor exception group priority value is lower than the current execution priority, or DEMCR.SDME == 0 and the DebugMonitor exception was generated in Secure state, then:

- The PE escalates a DebugMonitor synchronous exception that is generated by executing a `BKPT` instruction to a HardFault.
- The PE might set DEMCR.MON_PEND to 1 for a watchpoint debug event.
- The PE ignores the other debug events.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M** && **UDE**.*

$R_{CHXQ}$ A debug event that sets DEMCR.MON_PEND to 1 pends a DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{VSPX}$ DEMCR.MON_PEND is cleared to 0 when the PE takes a DebugMonitor exception. This means that a value of 1 for DEMCR.MON_PEND might never be observed for a synchronous DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BRXT}$ DFSR is updated at the same time as the PE sets DEMCR.MON_PEND to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BKHP}$ If an instruction that is being stepped or that generates a debug event reads DFSR or DEMCR, the value that is read for the relevant DFSR bit or for DEMCR.MON_PEND is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{VFLQ}$ For asynchronous debug events, if taken as a DebugMonitor exception, and if the current priority is lower than the DebugMonitor exception group priority, a DebugMonitor exception is taken in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{JVSC}$ A direct write to DEMCR can set DEMCR.MON_PEND to 1 at any time to make the DebugMonitor exception pending or can set DEMCR.MON_PEND to 0 to remove a pending DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{XPBN}$ When DEMCR.MON_PEND == 1, the PE takes the DebugMonitor exception according to the exception prioritization rules, regardless of the value of DEMCR.SDME and DEMCR.MON_EN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{NVQT}$ From Armv8.1 when DEMCR.MON_PEND == 1, the PE takes the DebugMonitor exception according to the exception prioritization rules, regardless of the value of DEMCR.SDME, DEMCR.MON_EN and DEMCR.UMON_EN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M** && **UDE**.*

$R_{LNCJ}$ Unless otherwise stated, asynchronous DebugMonitor exceptions can only cause preemption at instruction boundaries.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{BXBX}$ Instructions subject to beatwise execution begin to drain when a DebugMonitor exception is pended, when DEMCR.MON_PEND == 1, and the DebugMonitor exception is taken, subject to priority, after the instructions subject to beatwise execution have drained.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{PJJD}$ DebugMonitor exceptions cannot cause instruction resume or instruction restart. However, if another exception preempts an execution-continuable instruction that also generates a watchpoint, the PE might take that exception during the instruction, or abandon the instruction to take the exception, and, after returning from the exception, tail-chain to the DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

See also:

B13.2.2 *DWT unit operation* on page 335.

B13.5.2 *FPB unit operation* on page 359.

B3.27 *Exceptions, instruction resume, or instruction restart* on page 126.

### B12.4.2 Debug stepping

$R_{HMCN}$ The Armv8-M architecture supports debug stepping in both Halting debug and for the DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug** || **M**. Note, might require the DebugMonitor exception.*

$R_{THTG}$ It is IMPLEMENTATION DEFINED whether stepping a `WFE` or `WFI` instruction causes the `WFE` or `WFI` instruction to:

- Retire and take the debug event.
- Go into a sleep state and take the debug event only when another wake up event occurs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug** || **M**.*

$R_{LLVC}$ If a debug event wakes a `WFE` or `WFI` instruction, then on taking the debug event, the instruction has retired.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug** || **M**.*

$I_{CMYW}$ The debug architecture includes support for stepping over vector instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

Halting debug stepping.

Debug monitor stepping.

#### Halting debug stepping

$I_{QMXC}$ A debugger can use Halting debug stepping to exit from Debug state, execute a single instruction, and then reenter Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{SWKC}$ Halting debug stepping is active when all of the following apply:

- DHCSR.C_DEBUGEN is set to 1, Halting debug is enabled, and halting is allowed.
- DHCSR.C_STEP is set to 1, halting stepping is enabled.
- The PE is in Non-debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{ZVKS}$ When the PE exits Debug state and Halting debug stepping becomes active, the PE performs a Halting debug step as follows:

1. Performs one of the following:
   - Completes the next instruction without generating any exception.
   - Takes any pending exception entry of sufficient priority, without completing the next instruction. The PE performs an exception entry sequence that stacks the next instruction context. This context might include instruction continuation bits if the next instruction was partly executed and supports instruction resume. The exception might be a pending exception, or an exception generated by the execution of the next instruction.
   - Completes the execution of the next instruction, and then takes any pending exception of sufficient priority. The PE performs an exception entry sequence that stacks the following instruction context.
   - If the next instruction is an exception return instruction, completes the next instruction, tail-chaining to enter a new exception handler.

   In each case where the PE performs an exception entry sequence it does so according to the exception priority and late-arrival rules, meaning derived and late-arriving exceptions might preempt the exception entry sequence.

   The exception behavior is not recursive. Only a single `PushStack()` update can occur in a step sequence.
2. Sets DFSR.HALTED and DHCSR.C_HALT to 1. A read of the DFSR.HALTED or the DHCSR.C_HALT bit performed by the stepped instruction returns an UNKNOWN value.
3. After the Halting debug step, before executing the following instruction, because DHCSR.C_HALT is set the PE will halt and enter Debug state if halting is still allowed. However, if halting is prohibited after the Halting debug step then the PE does not enter Debug state and DHCSR.C_HALT remains set.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{LJJB}$ If the implementation includes the Armv8.1-M profile Vector Extension, a Halting debug step has the following additional possible actions when stepping over vector instructions. The PE:

- Attempts to complete the execution of all in-flight vector instructions, does not execute any new instructions, and generates a synchronous exception that is taken.
- Completes the execution of all in-flight vector instructions, does not start the execution of new instructions, and does not generate an exception.
- Completely executes the next vector instruction, without overlapping execution, and does not generate an exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && MVE**.*

$I_{LTRX}$ The debugger can optionally set DHCSR.C_MASKINTS to 1 to prevent PENDSV, SysTick, and external configurable interrupts from being taken. If a permitted exception becomes active, the PE steps into the exception handler and halts before executing the first instruction of the associated exception handler.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{ZDYR}$ If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, DHCSR.C_MASKINTS is ignored and has an Effective value 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{FWSN}$ If DHCSR.S_SDE == 0, DHCSR.C_MASKINTS is ignored for exceptions targeting Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{WCCK}$ If UDE is implemented and either DHCSR.S_SDE == 0, or DHCSR.S_SUIDE == 1, DHCSR.C_MASKINTS is ignored for exceptions targeting Secure state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**.*

$R_{LCXR}$ If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE or DHCSR.S_NSUIDE == 1, DHCSR.C_MASKINTS is ignored and has an Effective value of 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**.*

$R_{MBCB}$ DHCSR.{C_HALT, C_STEP, C_MASKINTS} can be written in a single write to DHCSR, as follows:

**DHCSR write**
assumes that DHCSR.C_DEBUGEN and DHCSR.S_HALT are both set to 1 when the write occurs and the PE is halted.

| C_HALT | C_STEP | C_MASKINTS | Effect |
|---|---|---|---|
| 0 | 0 | 0 | Exit Debug state and start instruction execution. Exceptions can become active and prioritized according to the priority rules and the configuration of exceptions. |
| 0 | 0 | 1 | Exit Debug state and start instruction execution. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active and proritized according to the priority rules. |
| 0 | 1 | 0 | Exit Debug state, step an instruction and halt. Exceptions can become active and prioritized according to the priority rules. |
| 0 | 1 | 1 | Exit Debug state, step an instruction and halt. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active and proritized according to the priority rules. |
| 1 | X | X | Remain in Debug state. |

The write to DHCSR assumes that DHCSR.C_DEBUGEN and DHCSR.S_HALT are both set to 1 when the write occurs, meaning the PE is halted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

## Debug monitor stepping

$I_{DXCT}$ A debugger can use debug monitor stepping to return from the DebugMonitor exception handler, execute a single instruction, and then reenter the DebugMonitor exception handler.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DebugMonitor exception**.*

$R_{MLRM}$ Debug monitor stepping is active when all of the following apply:

- DHCSR.C_DEBUGEN is set to 0 or the PE is in a state in which halting is prohibited.
- DEMCR.MON_EN is set to 1, that is Monitor debug is enabled.
- DEMCR.MON_STEP is set to 1, that is monitor stepping is enabled.
- DEMCR.SDME == 1 or the PE is in Non-secure state.
- Execution priority is below the priority of the DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{MJCF}$ From Armv8.1-M Debug monitor stepping is active when all of the following apply:

- DHCSR.C_DEBUGEN is set to 0 or the PE is in a state in which halting is prohibited.
- DEMCR.MON_EN is set to 1, or DEMCR.UMON_EN is set to 1 and the instruction is executed in unprivileged mode, that is Monitor debug is enabled.
- DEMCR.MON_STEP is set to 1, that is monitor stepping is enabled.
- DEMCR.SDME == 1 or the PE is in Non-secure state.
- Execution priority is below the priority of the DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M && UDE**.*

$R_{MWFT}$ When DebugMonitor stepping becomes active, the PE performs a DebugMonitor step as follows:

1. It performs one of the following:

- It completes the next instruction without generating any exception.
- It takes any pending exception of sufficient priority. The PE performs an exception entry sequence that stacks the next instruction context. The exception might be a pending exception, or it might be an exception generated by the execution of the next instruction.
- If the next instruction is an exception return instruction, the PE completes the next instruction, tail-chaining to enter a new exception handler according to the normal exception priority and late-arrival rules.

If the PE performs an exception entry sequence as part of step 1, the PE stacks the next instruction context. This context might include instruction continuation bits if the next instruction was partly executed and supports instruction resume.

2. If the execution priority is below the priority of the DebugMonitor exception after step 1, the PE sets DEMCR.MON_PEND and DFSR.HALTED to 1. A read of DEMCR.MON_PEND or DFSR.HALTED by the stepped instruction returns an UNKNOWN value.
3. Before executing the following instruction, the PE takes any pending exception with sufficient priority.

   If step 2 set DEMCR.MON_PEND to 1, then the DebugMonitor exception is pending. However, it is UNPREDICTABLE whether the PE uses the new value or the old value of DEMCR.MON_PEND in determining the highest priority exception. This means that:

   - Another exception might preempt execution before the DebugMonitor exception is taken, and the exception might be lower priority than the DebugMonitor exception. However, this is a Context synchronization event and the PE uses the new value of DEMCR.MON_PEND to determine the highest priority exception before executing the next instruction.
   - If no other exceptions are pending, the PE takes the DebugMonitor exception.

Derived and late-arriving exceptions might preempt the exception entry sequence.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{JKJM}$ If the implementation includes the Armv8.1-M profile Vector Extension, a Debug monitor step has the following additional possible actions when stepping over vector instructions. The PE:

- Attempts to complete the execution of all in-flight vector instructions, does not execute any new instructions, and generates a synchronous exception that is taken.
- Completes the execution of all in-flight vector instructions, does not start the execution of new instructions, and does not generate an exception.
- Completely executes the next vector instruction, without overlapping execution, and does not generate an exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M** && **MVE**.*

$I_{GPSX}$ In all other cases, the DebugMonitor exception preempting execution returns control to the DebugMonitor exception handler. Unless that handler clears DEMCR.MON_STEP to 0, returning from the handler performs the next debug monitor step.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$I_{KPKX}$ If, after the debug monitor stepping process, the taking of an exception means that the execution priority is no longer below that of the DebugMonitor exception, the values of DEMCR.MON_STEP and DEMCR.MON_PEND mean that debug monitor stepping process continues when execution priority falls back below the priority of the DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

### B12.4.3 Vector catch

$I_{TVRX}$ Vector catch is the mechanism for generating a debug event and entering Debug state on entry to a particular exception handler or reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{JCXR}$ Vector catching is only supported by Halting debug.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{PBVX}$ The conditions for a vector catch, other than reset vector catch, are:

- DHCSR.C_DEBUGEN == 1 and halting is allowed for the Security state the exception is targeting.
- The associated exception enable bit is set.
- The associated active bit is set.
- The associated vector catch enable bit.
- An exception is taken to the relevant exception handler. The associated fault status register status bit is set to 1.

When these conditions are met, the PE sets DHCSR.C_HALT to 1 and enters Debug state before executing the first instruction of the exception handler.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, If the Main Extension is not implemented only bits [24],[10] and [0] of **DEMCR** are implemented with Halting debug functionality. SecureFault requires S.*

$I_{XDGP}$ Late arrival and derived exceptions might occur, preempting the exception targeted by the vector catch and postponing when the PE halts.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{XKMH}$ The following table defines the exception, Fault status bit, and Vector catch bit.

| Exception | Fault status bit | Vector catch bit DEMCR |
|---|---|---|
| HardFault | HFSR.VECTTBL | VC_INTERR |
| | HFSR.FORCED | VC_HARDERR |
| | HFSR.DEBUGEVT | VC_HARDERR |
| BusFault | BFSR.IBUSERR | VC_BUSERR |
| | BFSR.PRECISERR | VC_BUSERR |
| | BFSR.IMPRECISERR | VC_BUSERR |
| | BFSR.UNSTKERR | VC_INTERR |
| | BFSR.STKERR | VC_INTERR |
| | BFSR.LSPERR | VC_INTERR |
| DebugMonitor | HFSR.DEBUGEVT | - |
| MemManage fault | MMFSR.IACCVIOL | VC_MMERR |
| | MMFSR.DACCVIOL | VC_MMERR |
| | MMFSR.MUNSTKERR | VC_INTERR |
| | MMFSR.MSTKERR | VC_INTERR |
| | MMFSR.MLSPERR | VC_INTERR |
| NMI | - | - |
| PENDSV | - | - |
| UsageFault | UFSR.UNDEFINSTR | VC_STATERR |
| | UFSR.INVSTATE | VC_STATERR |
| | UFSR.INVPC | VC_STATERR |
| | UFSR.NOCP | VC_NOCPERR |
| | UFSR.STKOF | VC_INTERR |
| | UFSR.UNALIGNED | VC_CHKERR |
| | UFSR.DIVBYZERO | VC_CHKERR |
| SecureFault | SFSR.INVEP | VC_SFERR |
| | SFSR.INVIS | VC_SFERR |
| | SFSR.INVER | VC_SFERR |
| | SFSR.AUVIOL | VC_SFERR |
| | SFSR.INVTRAN | VC_SFERR |
| | SFSR.LSPERR | VC_SFERR |
| | SFSR.LSERR | VC_SFERR |
| SVCall | - | - |
| SysTick | - | - |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{LKNL}$ When DHCSR.C_DEBUGEN == 0 or the PE is in a state in which halting is prohibited, all DEMCR.VC_ bits, other than DEMCR.VC_CORERESET, are ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{VZDB}$ If the PE resets into Secure state when both of:

- DHCSR.C_DEBUGEN == 1
- DEMCR.VC_CORERESET == 1

The PE will pend a Halt request and will halt and enter into Debug state when halting is permitted, setting DFSR.VCATCH to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{WRMQ}$ The PE pends a Vector catch event when all of the following is true:

- The PE has reset into Secure state.
- DHCSR.C_DEBUGEN == 1.
- DEMCR.VC_CORERESET == 1.
- Halting debug is not allowed in Secure state.

The PE does not halt until either it enters Non-secure state or debug is allowed in Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug && S**.*

$R_{MSWM}$ If UDE is implemented and all of the following are true:

- DHCSR.C_DEBUGEN == 1,
- DEMCR.VC_CORERESET == 1,
- `UnprivHaltingDebugAllowed(FALSE)` returns TRUE,
- The PE resets into Secure state,

The PE will pend a Vector Catch debug event, if debug is prohibited in Secure state, and does not halt until either the PE enters Non-secure state or debug is permitted in Secure state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**.*

See also:

B1.1 *Resets, Cold reset, and Warm reset* on page 61.

B3.10 *Exception enable, pending, and active bits* on page 83.

B3.13 *Priority model* on page 91.

B3.12 *Faults* on page 87.

B3.9 *Exception numbers and exception priority numbers* on page 80.

B3.24 *Exceptions during exception entry* on page 120.

B3.25 *Exceptions during exception return* on page 122.

Chapter B1 *Resets* on page 60.

### B12.4.4 Breakpoint instructions

$R_{CRJG}$ When DHCSR.C_DEBUGEN == 0 or when the PE is in a state in which halting is prohibited, the `BKPT` instruction does not generate an entry to Debug state. If no DebugMonitor exception is generated, the `BKPT` instruction generates a HardFault exception or enters Lockup state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MFHN}$ A `BKPT` instruction halts the PE if all of the following conditions apply:

- `HaltingDebugAllowed()` == TRUE.
- DHCSR.C_DEBUGEN == 1.
- The Security Extension is not implemented, the instruction is executed in Non-secure state, or DHCSR.S_SDE == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{WCQZ}$ If UDE is implemented a `BKPT` instruction halts the PE if all of the following conditions apply:

- `HaltingDebugAllowed()` == TRUE or `UnprivHaltingDebugAllowed()` == TRUE
- The Security Extension is not implemented, the instruction is executed in Non-secure state, or DHCSR.S_SDE

== 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**.*

$R_{FLKK}$ A BKPT instruction generates a DebugMonitor exception if it does not halt the PE and all of the following conditions apply:

- DEMCR.MON_EN == 1.
- The DebugMonitor exception group priority is greater than the current execution priority.
- The Security Extension is not implemented, the instruction is executed in Non-secure state, or DEMCR.SDME == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{ZJLD}$ If UDE is implemented a BKPT instruction generates a DebugMonitor exception if it does not halt the PE and either of the following conditions apply:

- DEMCR.MON_EN == 1
  - The DebugMonitor exception group priority is greater than the current execution priority.
  - The Security Extension is not implemented, the instruction is executed in Non-secure state, or DEMCR.SDME == 1.
- DEMCR.UMON_EN == 1 and the PE is executing in unprivileged mode and the all of the following conditions apply:
  - The DebugMonitor exception group priority is greater than the current execution priority.
  - The Security Extension is not implemented, the instruction is executed in Non-secure state, or DEMCR.SDME == 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M && UDE**.*

### B12.4.5 External debug request

$R_{XZCP}$ When the PE is in Non-debug state, an external agent can signal an external debug request.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GTGX}$ An external debug request can cause a debug event, that causes either:

- Entry to Debug state.
- If the Main Extension is implemented, a DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M** || **Halting debug**.*

$R_{FGCV}$ The PE ignores external debug requests when it is in Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{BXRD}$ When DHCSR.C_DEBUGEN == 0 or the PE is in a state in which halting is prohibited, an External debug request does not generate an entry to Debug state and is ignored if no DebugMonitor exception is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WGMB}$ If the DebugMonitor exception group priority is greater than the current execution priority and DEMCR.MON_EN == 1, an External debug request that does not generate an entry to Debug state sets DEMCR.MON_PEND to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

$R_{BNBR}$ If the DebugMonitor exception group priority is greater than the current execution priority and DEMCR.MON_EN == 1, or DEMCR.UMON_EN == 1 and the PE is executing in unprivileged mode, an External debug request that does not generate an entry to Debug state sets DEMCR.MON_PEND to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **M && UDE**.*

See also:

B12.4 *Debug event behavior* on page 302.

DFSR.EXTERNAL.

## B12.5 Debug state

$R_{RMKS}$ In Halting debug, debug events allow an external debugger to halt the PE. The PE then enters Debug state. When the PE is in Debug state:

- The PE stops executing instructions from the location indicated by the PC, and is instead controlled by the external debug interface.
- The PE cannot service any interrupts.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{QDCP}$ In Debug state, the PE clears the DHCSR.S_REGRDY bit to 0 when the debugger writes to DCRSR and the PE then sets the bit to 1 when the transfer between the DCRDR and R0-R12 (Rn), Special-purpose register, Floating-point Extension register, or DebugReturnAddress completes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**. Note, Floating-point registers are* RES0 *if FP is not implemented.*

$I_{FKSM}$ To transfer a word to a general-purpose register, to a Special-purpose register, to a Floating-point Extension register, or to DebugReturnAddress, a debugger:

1. Writes the required word to DCRDR.
2. Writes to the DCRSR, with the REGSEL value indicating the required register, and the REGWnR bit set to 1 to indicate a write access. This clears the DHCSR.S_REGRDY bit to 0.
3. If required, polls DHCSR until DHCSR.S_REGRDY reads-as-one. This shows that the PE has transferred the DCRDR value to the selected register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{CMBB}$ To transfer a word from a general-purpose register, from a Special-purpose register, from a Floating-point Extension register, or from DebugReturnAddress, a debugger:

1. Writes to DCRSR, with the REGSEL value indicating the required register, and the REGWnR bit as 0 to indicate a read access. This clears the DHCSR.S_REGRDY bit to 0.
2. Polls DHCSR until DHCSR.S_REGRDY reads-as-one. This shows that the PE has transferred the value of the selected register to DCRDR.
3. Reads the required value from DCRDR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{VLVD}$ In Debug state, following a write to DCRSR that clears the DHCSR.S_REGRDY bit to 0, the behavior is UNPREDICTABLE if any of the following occur before the PE sets DHCSR.S_REGRDY to 1:

- The PE exits Debug state, other than because of a Warm reset.
- The debugger writes to DCRDR or DCRSR.

If the DCRSR.REGWnR bit was set to 0 and the debugger reads from DCRDR before the PE sets DHCSR.S_REGRDY to 1, then the read returns an UNKNOWN value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{JKBB}$ When using the DCRDR, DCRSR and DHCSR.S_REGRDY mechanism to write to XPSR, all bits of the XPSR are fully accessible. The effect of writing an illegal value is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{RXQB}$ The DCRDR, DCRSR and DHCSR.S_REGRDY mechanism differs from the behavior of MSR or MRS instruction accesses to the XPSR, where some bits are ignored on writes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{QLRN}$ When the PE is halted the Debugger can write to:

- The DebugReturnAddress.
- EPSR.IT/ICI bits.

On exiting Debug state the PE starts from DebugReturnAddress. The Debugger must ensure that the EPSR.IT and EPSR.ICI bits are consistent with DebugReturnAddress, otherwise instruction execution will be UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{XZJN}$ The debugger can write to the EPSR.IT/ICI/ECI bits. If the debugger does this, it writes a value consistent with the instruction to be executed on exiting Debug state, otherwise instruction execution will be UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug** && **MVE**.*

$I_{RRFN}$ The debugger can always set FAULTMASK to 1, but doing so might cause unexpected behavior on exit from Debug state. An `MSR` instruction cannot set FAULTMASK to 1 when the execution priority is -1 or higher.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{XRRQ}$ The debugger can write to the EPSR.IT/ICI bits, and on exiting Debug state any interrupted `LDM` or `STM` instruction will use these new values. Clearing the ICI bits to 0 will cause the interrupted `LDM` or `STM` instruction to restart or continue.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{QLTB}$ The debugger can write to the EPSR.ECI bits, and on exiting Debug state any interrupted vector instructions will use these new values. Clearing the ECI bits to 0 will cause the interrupted vector instruction to restart.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug** && **MVE**.*

$R_{BMHD}$ When the PE is in Debug state, an indirect write to a Special-purpose register caused by an access by the debugger to a register within the System Control Block (SCB) is guaranteed to be visible after the access to the register within the SCB completed to a subsequent:

- Access to the Special-purpose register through DCRDR.
- Indirect read of the Special-purpose register made for an access of any register through DCRDR or any register within the System Control Block.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{MDJX}$ When the PE is in Debug state, a write to a Special-purpose register made by the debugger through the DCRDR is guaranteed to be visible after the write is observed to be completed in DHCSR.S_REGRDY to a subsequent:

- Access of any register through DCRDR or any register within the System Control Block.
- Indirect read of the Special-purpose register made for an access to any register through DCRDR or any register within the System Control Block.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{DMTG}$ A read or write of a register through DCRDR starts with a write to DCRSR. Where the architecture guarantees that a previous access is visible to a subsequent access through DCRDR, this means the write to DCRSR is made after the point where the previous access is visible.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{WFHL}$ Armv8.1-M introduces restrictions on the DCRDR and DCRSR mechanism for unprivileged debug access to the floating-point registers when lazy state preservation is active or CPACR.CP10 associated with the Security state of the floating-point context restricts access.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**.*

$R_{HTXM}$ An access using the DCRDR and DCRSR mechanism to the FPSCR, VPR or the floating-point register file will return RAZ/WI if `CanDebugAccessFP()` returns FALSE. When `CanDebugAccessFP()` returns TRUE, DHCSR.S_FPD is set to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **Halting debug && UDE**. Note, Floating-point registers are* RES0 *if FP is not implemented.*

See also:

*DCRDR, Debug Core Register Data Register*.

*DCRSR, Debug Core Data Select Register*.

`DebugRegisterTransfer()`

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

## B12.6 Exiting Debug state

$R_{BFGT}$ The PE exits Debug state:

- When the debugger writes 0 to DHCSR.C_HALT.
- On receipt of an external restart request.
- On Warm reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{GGMJ}$ For synchronous debug events DebugReturnAddress is:

| Synchronous debug event | DebugReturnAddress |
|---|---|
| Breakpoint debug events (BKPT or FPB Match) | Address of the breakpointed instruction. |
| Vector Catch debug events | Address of the first instrucion of the exception handler. |
| Step debug events | Address of the next instruction to be executed in simple sequential execution order following the instruction that was stepped. If an exception was taken during stepping, this is the first instruction of the exception handler. |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{XCCB}$ Bit[0] of a DebugReturnAddress value is RAZ/SBZ. When writing a DebugReturnAddress, writing bit [0] of the address does not affect the EPSR.T bit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{HNKB}$ Exiting Debug state has no architecturally defined effect on the Event Register and exclusive monitors.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{WKSD}$ If software clears DHCSR.C_HALT to 0 when the PE is in Debug state, a subsequent read of the DHCSR that returns 1 for both DHCSR.C_HALT and DHCSR.S_HALT indicates that the PE has reentered Debug state because it has detected a new debug event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{FKXH}$ Before leaving Debug state caused by an imprecise entry into Debug state the system is reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

See also:

B12.5 *Debug state* on page 318

## B12.7 Multiprocessor support

$R_{QXLS}$ Systems that support debug of more than one PE, either within a single device or as heterogeneous PEs in a more complex system, require each PE to support all of the following to enable cross-triggering of debug events between PEs:

- An external debug request.
- A cross-halt event.
- An external restart request.

Support for these features is OPTIONAL in other systems.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### B12.7.1 Cross-halt event

$R_{DLCV}$ When the PE enters Debug state, it signals to an external agent that it is entering Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

### B12.7.2 External restart request

$R_{ZKVW}$ When the PE is in Debug state, an external agent can signal an external restart request that causes the PE to exit Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$R_{WJST}$ An external restart request is not ordered with respect to accesses to memory-mapped registers. It is UNPREDICTABLE whether an access to a memory-mapped register from a DAP completes before an external restart request.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **Halting debug**.*

$I_{VNDK}$ A debugger ensures that any read or write of a memory-mapped register by the DAP completes before issuing an external restart request.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DB**.*

$R_{NJQN}$ The PE ignores external restart requests when it is in Non-debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B12.6 *Exiting Debug state* on page 322.

# Chapter B13
# Debug and Trace Components

This chapter specifies the Armv8-M debug and trace component rules. It contains the following sections:

# B13.1 Instrumentation Trace Macrocell

## B13.1.1 About the ITM

$R_{GDNG}$ The *Instrumentation Trace Macrocell* (ITM) provides a memory-mapped register interface that applications can use to generate Instrumentation packets.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$I_{BXWJ}$ The ITM is only available if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{LMXS}$ The ITM generates Instrumentation packets, Synchronization packets, and the following protocol packets:

- Overflow packets.
- Local timestamp packets.
- Global timestamp packets.
- Extension packets.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XQRX}$ The ITM combines the following packets into a single trace stream:

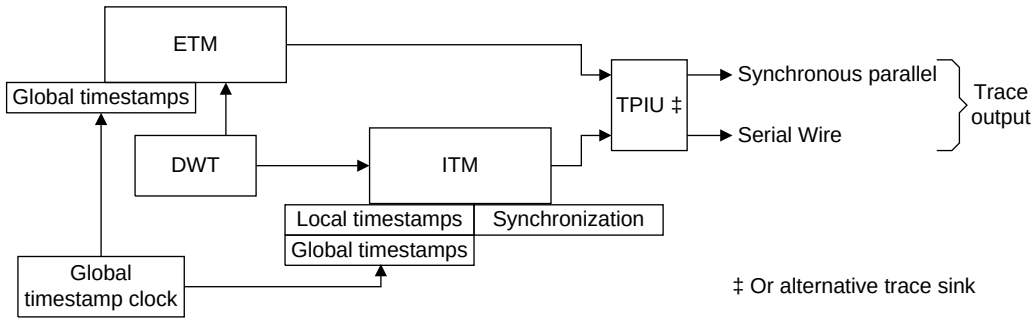
- Instrumentation packets.
- Synchronization packets.
- Protocol packets.
- Hardware source packets that are generated by the DWT.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{MXMN}$ If the implementation includes the PMU, the PMU Hardware source packets are included in the single trace stream.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **ITM** && **PMU**.*

$I_{FQLR}$ The following figure shows how the ITM relates to other debug components.




*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{BWJJ}$ When multiple sources are generating data at the same time, the ITM arbitrates using the following priorities:

**Synchronization, when required**: Priority level -1, highest.

**Instrumentation**: Priority level 0.

**Hardware source**: Priority level 1.

**Local timestamps**: Priority level 2.

**Global timestamp 1**: Priority level 3.

**Global timestamp 2**: Priority level 4.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

See also:

Global timestamping.

B13.2 *Data Watchpoint and Trace unit* on page 334.

ITM and DWT Packet Protocol Specification.

### B13.1.2 ITM operation

$R_{NKSC}$ The ITM consists of:

- Up to 256 stimulus port registers, ITM_STIMn.
- Up to eight enable registers, ITM_TERn.
- An access control register, ITM_TPR.
- A general control register, ITM_TCR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{MFDV}$ The number of ITM_STIMn registers is an IMPLEMENTATION DEFINED multiple of eight. Software can discover the number of supported stimulus ports by writing all ones to the ITM_TPR, and then reading how many bits are set to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{CGVD}$ If the ITM is disabled or not implemented, any Secure or Non-secure write to ITM_STIMn is ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM** && **S**.*

$R_{NJTR}$ Unprivileged and privileged software can always read all ITM registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{FFXF}$ If the ITM is not implemented, the ITM registers are RAZ/WI.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{CSFV}$ The ITM_TPR defines whether each group of eight ITM_STIMn registers, and their corresponding ITM_TERn bits, can be written by an unprivileged access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{PTXV}$ ITM_STIMn registers are 32-bit registers that support the following word-aligned accesses:

- Byte accesses, to access register bits[7:0].
- Halfword accesses, to access register bits[15:0].
- Word accesses, to access register bits[31:0].

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{LNMW}$ Non-word-aligned accesses are UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{NQVK}$ ITM_TCR.ITMENA is a global enable bit for the ITM. A Cold reset clears this bit to 0, disabling the ITM.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{VRGP}$ The ITM_TERn registers provide an enable bit for each stimulus port.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{NTCR}$ When software writes to an enabled ITM_STIMn register, the ITM combines the identity of the port, the size of the write access, and the data that is written, into an Instrumentation packet that it writes to a stimulus port output buffer. The ITM transmits packetsfrom the output buffer to a trace sink.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{TCTH}$ If DEMCR.TRCENA == 0 or `NoninvasiveDebugAllowed()` == FALSE, the ITM does not generate trace.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{GRNM}$ The size of the stimulus port output buffer is IMPLEMENTATION DEFINED, but has at least one entry. The stimulus port output buffer is shared by all ITM_STIMn registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{SXNK}$ When the stimulus port output buffer is full, if software writes to any ITM_STIMn register, the ITM discards the write data, and generates an Overflow packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{SRPP}$ Reading the ITM_STIMn register of any enabled stimulus port returns a value indicating the output buffer status and that the port is enabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XVVB}$ Reading an ITM_STIMn register when the ITM is disabled, or when the individual stimulus port is disabled in the corresponding ITM_TERn register, returns the value indicating that the output buffer cannot accept data because the port is disabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{FXSL}$ Hardware source packets that are generated by any source use a separate output buffer. The output buffer status that is obtained by reading an ITM_STIMn register is not affected by trace that is generated by any other source.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM && DWT-T**.*

$R_{RGCV}$ Stalling is supported through an optional control, ITM_TCR.STALLENA. When implemented and set to 1, the ITM can stall the PE to guarantee delivery of the following Hardware source packets:

- Data Trace Data Address.
- Data Trace Data Value.
- Data Trace Match.
- Data Trace PC Value.
- Exception Trace.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{NFJN}$ Stalling does not affect the DWT counters.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM && DWT-T**.*

$R_{NWVT}$ Stalling does not affect the PMU counters.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **ITM && PMU**.*

$R_{TNDP}$ The ITM might generate an Overflow packet while the PE is stalled, if the DWT generates:

- A Hardware source packet other than a Data trace packet or Exception packet.
- A Data Trace PC value packet or Data Trace Match packet from a Cycle Counter comparator.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{CRKK}$ The ITM does not stall the PE in Secure state if `SecureHaltingDebugAllowed()` == FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM** && **S**.*

$R_{GRHW}$ The ITM does not stall the PE if `HaltingDebugAllowed()` == FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{BGCP}$ The ITM does not stall the PE in such a way as to deadlock the system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{FRJG}$ The ITM does not stall the PE if the trace output is disabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XRVL}$ The ITM does not stall for writes to the ITM_STIMn registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{HDLH}$ Instrumentation trace packets appear in the trace output in the order in which writes arrive at the ITM_STIMn registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XNHX}$ It is IMPLEMENTATION DEFINED whether an ITM requires flushing of trace data to guarantee that data is output.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{TSXR}$ If periodic flushing is required, the ITM flushes trace data:

- When a Synchronization packet is generated.
- When trace is disabled, meaning that either DEMCR.TRCENA is cleared to 0 or one or more of ITM_TCR.{TXENA, SYNCENA, TSENA, SYNCENA} is cleared to 0, and the buffered trace includes at least one corresponding packet type.
- In response to other IMPLEMENTATION DEFINED flush requests from the system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{MKFS}$ If a system supports multiple trace streams, the debugger writes a unique nonzero trace ID value to the ITM_TCR.TraceBusID field. The system uses this value to identify the individual trace streams. To avoid trace stream corruption, before modifying the ITM_TCR.TraceBusID a debugger does the following:

- It clears the ITM_TCR.ITMENA bit to 0, to disable the ITM.
- It polls the ITM_TCR.BUSY bit until it returns to 0, indicating that the ITM is inactive.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

### B13.1.3 Timestamp support

$R_{RVLT}$ Timestamps provide information on the timing of event generation regarding their visibility at a trace output port.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{TFDG}$ An Armv8-M PE can implement either or both of the following types of timestamp:

- Local timestamps.
- Global timestamps.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

### Local timestamping

$R_{RMXM}$ Local timestamps provide delta timestamp values, meaning each local timestamp indicates the elapsed time since generating the previous local timestamp.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{WGBG}$ The ITM generates the local timestamps from the timestamp counter in the ITM unit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XLBH}$ The timestamp counter size is an IMPLEMENTATION DEFINED value that is less than or equal to 28 bits.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{GPXT}$ It is IMPLEMENTATION DEFINED whether the ITM supports synchronous clocking of the timestamp counter mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{SRJH}$ It is IMPLEMENTATION DEFINED whether the ITM and TPIU support asynchronous clocking of the timestamp counter mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{GHPS}$ ITM_TCR.TSENA enables Local timestamp packet generation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{FSWG}$ When local timestamping is enabled and the DWT or ITM transfers a Hardware source or instrumentation trace packet to the appropriate output FIFO, and the timestamp counter is nonzero, the ITM:

- Generates a Local timestamp packet.
- Resets the timestamp counter to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{BRRL}$ If the timestamp counter overflows, it continues counting from zero and the ITM generates an Overflow packet and transmits an associated Local timestamp packet at the earliest opportunity. If higher priority trace packets delay transmission of this Local timestamp packet, the timestamp packet has the appropriate nonzero local timestamp value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XFRH}$ The ITM can generate a Local timestamp packet relating to a single event packet, or to a stream of back-to-back packets if multiple events generate a packet stream without any idle time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{QJJB}$ Local timestamp packets include status information that indicates any delay in one or both of:

- Transmission of the timestamp packet relative to the corresponding event packet.
- Transmission of the corresponding event packet relative to the event itself.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{NDCK}$ If the ITM cannot generate a Local timestamp packet synchronously with the corresponding event packet, the timestamp count continues to increment until the ITM can generate a Local timestamp packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{TBMX}$ The ITM compresses the count value in the timestamp packet by removing leading zeroes, and transmits the smallest packet that can hold the required count value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$I_{SQLG}$ To prevent overflow, Arm recommends that the ITM emits a Local timestamp packet before the timestamp counter overflows.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

### Local timestamp clocking options

$R_{DSTG}$ If the implementation supports both synchronous and asynchronous clocking of the local timestamp counter, ITM_TCR.SWOENA selects the clocking mode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{BDWS}$ When software selects synchronous clocking, when local timestamping is enabled, the PE clock drives the timestamp counter, and the counter increments on each PE clock cycle.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$I_{JQJD}$ When software selects synchronous clocking, whether local timestamps are generated in Debug state is IMPLEMENTATION DEFINED. Arm recommends that entering Debug state disables local timestamping, regardless of the value of the ITM_TCR.TSENA bit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{JDRD}$ When software selects asynchronous clocking, and enables local timestamping, the TPIU output interface clock drives the timestamp counter, through a configurable prescaler. The rate of asynchronous clocking depends on the output encoding scheme. This clock might be asynchronous to the PE clock.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{NGDW}$ When asynchronous clocking is implemented, whether the incoming clock signal can be divided before driving the local timestamping counter is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{RMTN}$ If the implementation supports division of the incoming asynchronous clock signal, ITM_TCR.TSPrescale sets the prescaler divide value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{SKCP}$ Software only selects asynchronous clocking when the TPIU is configured to use an output mode that supports asynchronous clocking.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM && TPIU**.*

$R_{JGCF}$ When software selects asynchronous clocking and the TPIU asynchronous interface is idle, the ITM holds the timestamp counter at zero. This means that the ITM does not generate a local timestamp on the first packet after an idle on the asynchronous interface.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM && TPIU**.*

See also:

B13.4 *Trace Port Interface Unit* on page 357.

### Global timestamping

$I_{DKSD}$ Global timestamps provide absolute timestamp values, which are based on a system global timestamp clock. They provide synchronization between different trace sources in the system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{HBWD}$ If an implementation includes Global timestamping, the ITM generates *Global timestamp* (GTS) packets, which are based on a global timestamp clock.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{KWQJ}$ The size of the global timestamp is either 48 bits or 64 bits. The choice between these two options is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{SRDF}$ To transfer the global timestamp, two formats of Global timestamp packets are defined:

- The first packet format, Global timestamp 1 packet, holds the value of the least significant timestamp bits[25:0], and wrap and clock change indicators.
- The second packet format, Global timestamp 2 packet, holds the value of the high-order timestamp bits:
  - Bits[47:26], if a 48-bit global timestamp is supported.
  - Bits[63:26], if a 64-bit global timestamp is supported.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{VGBT}$ The ITM generates a full Global timestamp packet, consisting of Global timestamp 1 packet Global timestamp 2 packet, in the following circumstances:

- When software first enables global timestamps, by changing the value of the ITM_TCR.GTSFREQ field from zero to a nonzero value.
- When the system asserts the clock ratio change signal in the external ITM timestamp interface.
- In response to a Synchronization packet request, even if ITM_TCR.SYNCENA == 0.
- When the ITM has to generate a global timestamp, and the ITM detects that the value of the high-order bits of the global timestamp have changed since the Global timestamp 2 packet was last generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XQWL}$ If the global timestamp generated by the ITM does not have to be a full global timestamp, the ITM generates only a single Global timestamp 1 packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{DJLN}$ When the ITM generates a global timestamp, it does so after a non-delayed Instrumentation or Hardware Source packet. The Global Timestamp 1 packet is always associated with the most recently output non-delayed Instrumentation or Hardware Source packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{WDCX}$ When the ITM generates a full global timestamp:

1. The ITM first generates the Global timestamp 1 packet with timestamp bits[25:0], with the applicable bit of the Wrap and ClockCh bits in that packet set to 1 to indicate that the high-order bits of the timestamp will also be output. This is the packet that the ITM outputs immediately after a non-delayed trace packet.
2. Because of packet prioritization, the ITM might have to transmit other trace packets before it can output the Global timestamp 2 packet that contains the high-order bits of the timestamp. It might also have to transmit another Global timestamp packet. If so, it outputs the Global timestamp 1 packet with timestamp bits[25:0] and the Wrap bit set to 1.

3. The ITM later generates the Global timestamp 2 packet with the high-order timestamp bits for the most recently transmitted Global timestamp 1 packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

See also:

B13.1.4 *Synchronization support* .

B13.1.5 *Continuation bits* .

ITM and DWT Packet Protocol Specification.

### B13.1.4 Synchronization support

$I_{LRJT}$ An external debugger uses Synchronization Packets to recover bit-to-byte alignment information in a serial data stream.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$I_{LVGD}$ Synchronization packets are independent of timestamp packets.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$I_{JNJV}$ Arm recommends that software disables Synchronization packets when using an asynchronous serial trace port, to reduce the data stream bandwidth.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{RMND}$ If ITM_TCR.SYNCENA == 1, the ITM outputs a Synchronization packet:

- When it is first enabled.
- If DWT_CYCCNT is implemented and DWT_CTRL.SYNCTAP is nonzero, in response to a Synchronization packet request from the DWT unit.
- If TPIU_PSCR is implemented, in response to a Synchronization packet request from the TPIU:
  - If DWT_CYCCNT is not implemented, TPIU_PSCR is implemented.
  - If DWT_CYCCNT is implemented, it is IMPLEMENTATION DEFINED whether TPIU_PSCR is implemented.
- In response to other IMPLEMENTATION DEFINED Synchronization packet requests from the system.
- On exit from Debug state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**. Note, might require additional extensions as described in the rule.*

See also:

DWT_CTRL.SYNCTAP.

### B13.1.5 Continuation bits

$I_{BFMX}$ A Synchronization packet consists of a bit stream of at least 47 zero bits followed by a one bit. The final bit is the byte alignment marker, and therefore bit[7] of the last byte of a Synchronization packet is always one.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{JNVH}$ The longest Extension packet is always 5 bytes. In an Extension packet, bit[7] of each byte, including the header byte, but not including the last byte of a 5-byte packet, is a continuation bit, C. Bit[7] of the last byte of a 5-byte Extension packet is part of the extension field. Bit[7] of the last byte of a fewer-than-5-byte Extension packet is always zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{XFTL}$ For all other protocol packets, bit[7] of each byte, including the header byte, but not including the last byte of a 7-byte packet, is a continuation bit, C. Bit[7] of the last byte of a packet is always zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{BBSF}$ Each packet type defines its maximum packet length. Except for Global timestamp 2 and Synchronization packets, the longest defined packet is 5 bytes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

$R_{DPJG}$ The continuation bit, C, is defined as:

**0**: This is the last byte of the packet.

**1**: This is not the last byte of the packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ITM**.*

# B13.2 Data Watchpoint and Trace unit

## B13.2.1 About the DWT

$R_{QQLQ}$ The *Data Watchpoint and Trace* (DWT) unit provides the following features:

- Comparators that support:
  - Use as a single comparator for instruction address matching or data address matching.
  - Use in linked pairs for instruction address range matching or data address range matching.
- Generation, on a comparator match, of:
  - A debug event that causes the PE either to enter Debug state or, if the Main Extension is implemented, to take a DebugMonitor exception.
  - Signaling a match to an ETM, if implemented.
  - Signaling a match to another external resource.
- External instruction address sampling using an instruction address sample register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && (DebugMonitor exception || Halting debug)**. Note, some comparator matches require ETM.*

$R_{KBMX}$ If the Main Extension is implemented, the DWT provides the following features:

- An optional cycle counter.
- Comparators that support:
  - Use as a single comparator for cycle counter matching, if the cycle counter is implemented.
  - Use as a single comparator for data value matching.
  - Use in linked pairs for data value matching at a specific data address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M**.*

$R_{DVJV}$ If the Main Extension and the ITM are implemented, the DWT provides the following trace generation features:

- Generating one or more trace packets on a comparator match.
- Generating periodic trace packets for software profiling.
- Exception trace.
- Performance profiling counters that generate trace.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M && ITM**.*

$R_{CPXJ}$ If DWT_CTRL.NOTRCPKT is 1, there is no DWT trace support.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FKFP}$ If DWT_CTRL.NOCYCCNT is 1, there is no cycle counter support.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{BKGF}$ If DWT_CTRL.NOPRFCNT is 1, there is no profiling counter support.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{HFTT}$ The DWT_CTRL.NUMCOMP field indicates the number of implemented DWT comparators, which is in the range 0-15.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VMSD}$ It is optional whether the DWT supports Data value masking. If Data value masking is supported, DWT_DEVARCH.REVISION==0b0001.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

$R_{WQLX}$ If the Main Extension is not implemented, Cycle counter, Data value, Linked data value, and Data address with value comparators and all trace features are not supported.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M && DWT-T**.*

$R_{SSWT}$ Data trace packets are only generated for comparators 0-3.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CRHX}$ When a DWT implementation includes one or more comparators, which comparator features are supported, and by which comparators, is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### B13.2.2 DWT unit operation

$I_{WTSS}$ For each implemented comparator, a set of registers defines the comparator operation. For comparator *n*:

- DWT_COMPn holds a value for the comparison.
- DWT_FUNCTIONn defines the operation of the comparator.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{LZMQ}$ DWT_VMASK<n> is an additional register that defines the comparator operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

$R_{XBRD}$ A *Secure match* is a match that is generated by one of the following:

- Vector fetches where NS-Req has a value of Secure for the operation.
- The hardware stacking or unstacking of registers, where NS-Req has a value of Secure for the operation, on any of:
  - Exception entry.
  - Exception exit.
  - Function call entry.
  - Function return.
  - Lazy state preservation.
- An operation that is generated by an instruction that is executed in Secure state, including:
  - An Instruction address match for an instruction that is executed in Secure state.
  - A Data address or Data value match for a load or store that is generated by an instruction that is executed in Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{DVCN}$ A Secure match can be generated by a cycle counter match in Secure state if DWT_CTRL.CYCDISS == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{MGGT}$ For a comparator <*n*>, all matches are prohibited if one or more of the following conditions apply:

- DEMCR.TRCENA == 0 or `NoninvasiveDebugAllowed()` == FALSE.
- DWT_FUNCTION.ACTION specifies a debug event and all the following conditions apply:

- HaltingDebugAllowed() == FALSE or DHCSR.C_DEBUGEN == 0.
- The Main Extension is not implemented or DEMCR.MON_EN == 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QCKL}$ For a comparator <*n*>, all matches are prohibited if one or more of the following conditions apply:

- DEMCR.TRCENA == 0 or NoninvasiveDebugAllowed() == FALSE.
- DWT_FUNCTION.ACTION specifies a debug event and all the following conditions apply:
  - HaltingDebugAllowed() == FALSE or DHCSR.C_DEBUGEN == 0.
  - The Main Extension is not implemented or DEMCR.MON_EN == 0 or DEMCR.UMON_EN == 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T**. Note, UDE required for DEMCR.UMON_EN.*

$R_{GFLN}$ Secure matches are prohibited for a comparator if one of the following conditions applies:

- DWT_FUNCTION.ACTION specifies a trace or trigger event and SecureNoninvasiveDebugAllowed() == FALSE.
- DWT_FUNCTION.ACTION specifies a debug event and all of the following conditions apply:
  - DHCSR.S_SDE == 0.
  - The Main Extension is not implemented or DEMCR.SDME == 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**. Note, M required if DEMCR.SDME == 1.*

$R_{HCFP}$ For address and value comparisons, the control register values and the current execution priority and Security state relate to the state of the PE when it generated the transaction that is being matched against.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{FFKV}$ Between a change to the debug authentication interface, DHCSR or DEMCR, that disables debug and a following context synchronization event, it is UNPREDICTABLE whether the DWT uses the old values or the new values.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VTNJ}$ Where the DWT operation rules prohibit a match being generated, a match is not generated, even if the programmers' model defines it as being UNPREDICTABLE whether a comparator generates a match as the result of the way in which the DWT is programmed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{PKRK}$ If DEMCR.TRCENA == 0 or NoninvasiveDebugAllowed() == FALSE, DWT_CTRL.FOLDEVTENA, LSUEVTENA, SLEEPEVTENA, EXCEVTENA, and CPIEVTENA are ignored, and these fields have an Effective value of 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GDMN}$ If DEMCR.TRCENA == 0 or NoninvasiveDebugAllowed() == FALSE, the DWT does not generate any trace packets.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FHWV}$ If SecureNoninvasiveDebugAllowed() == FALSE, DWT_CTRL.FOLDEVTENA, LSUEVTENA, SLEEPEVTENA, EXCEVTENA, and CPIEVTENA are ignored and these fields have an Effective value of 0 in Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{WSRR}$ If `SecureNoninvasiveDebugAllowed()` == FALSE, Exception trace packets are not generated if the exception number in the packet represents a Secure exception:

- Exception entry packets are not generated for exceptions that are taken to Secure state.
- Exception exit packets are not generated for exits from Secure state.
- Exception return packets are not generated for returns to Secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{DFWR}$ Exception trace packets appear in the same order as for a simple sequential execution of the exception handling.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{XDVS}$ The cycle counter, DWT_CYCCNT, and the POSTCNT counter are disabled when DEMCR.TRCENA == 0, but are not otherwise affected by debug authentication.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{RTJR}$ The cycle counter does not count in Secure state when DWT_CTRL.CYCDISS is set to 1. This is independent of Secure debug authentication.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{BRSR}$ When the DWT generates a match, DWT_FUNCTION.MATCHED is set to 1, unless the comparator is a Data address limit or Instruction address limit comparator, in which case DWT_FUNCTION.MATCHED is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NRGV}$ When the DWT generates a match, then if DWT_FUNCTION.ACTION specifies a debug event, then DHCSR.C_HALT is set to 1 if all of the following conditions are true:

- `HaltingDebugAllowed()` == TRUE.
- DHCSR.C_DEBUGEN == 1.
- DHCSR.S_HALT == 0.
- Either the match is not a Secure match or DHCSR.S_SDE == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{PJGW}$ When the DWT generates a match, then if DWT_FUNCTION.ACTION specifies a debug event, DEMCR.MON_PEND is set to 1 if all of the following conditions apply:

- `HaltingDebugAllowed()` == FALSE, DHCSR.C_DEBUGEN == 0, or the match is a Secure match and DHCSR.S_SDE == 0.
- DEMCR.MON_EN == 1.
- Either the DebugMonitor exception group priority is greater than the execution priority of the access and the watchpoint was not generated by a lazy state preservation access, or FPCCR.MONRDY has a value of 1 and the watchpoint was generated by lazy state preservation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M**.*

$R_{CQPJ}$ When the DWT generates a match, then if DWT_FUNCTION.ACTION specifies a debug event, DEMCR.MON_PEND is set to 1 if all of the following conditions apply:

- `HaltingDebugAllowed()` == FALSE, DHCSR.C_DEBUGEN == 0, or the match is a Secure match and DHCSR.S_SDE == 0.
- DEMCR.MON_EN == 1 or DEMCR.UMON_EN == 1 and if the privilege mode of the attributable Security state is unprivileged.
- Either the DebugMonitor exception group priority is greater than the current execution priority and the watchpoint was not generated by a lazy state preservation access, or FPCCR.MONRDY has a value of 1 and the watchpoint was generated by lazy state preservation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T && M && UDE**.*

$R_{FTBG}$ When the DWT generates a match, then a Data trace match packet is generated, if all of the following conditions apply:

- `SecureNoninvasiveDebugAllowed()` == FALSE.
- DWT_FUNCTION.ACTION specifies generating a Data trace PC value packet.
- The instruction address that would be included in the packet refers to an instruction that was executed in Secure state.

Otherwise, the type of trace packet that is specified by DWT_FUNCTION.ACTION is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M && S**.*

$R_{FNDW}$ An access that results in a MemManage fault or SecureFault exception because of the alignment, SAU, IDAU, or MPU checks, is not observed by the DWT, and cannot generate a match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && (S || M && MPU)**.*

$R_{PGJB}$ The DWT treats hardware accesses to the stack as data accesses:

- For registers pushed to the stack by hardware as part of an exception entry or lazy state preservation.
- For registers popped from the stack by hardware as part of an exception return.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NQNR}$ The DWT treats hardware accesses to the stack as data accesses:

- For registers pushed to the stack by hardware as part of a Non-secure function call.
- For registers popped from the stack by hardware as part of a Non-secure function.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && S**.*

$R_{SFSC}$ Where a hardware access to the stack generates a Data trace PC value packet, the PC value in the packet will be as follows:

- On exception entry or a function call, the PC value will be the return address for the exception or function call.
- On lazy state preservation the PC value is the address of the instruction that triggered the lazy state preservation.
- On exception return or Non-secure function return the PC value is either:
  - The address of the instruction that caused the exception return or the Non-secure function return.
  - The EXC_RETURN or FNC_RETURN payload value used in the exception return or the Non-secure function return.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{YZLM}$ Watchpoints that occur as a result of DWT Unit events will be taken when all in-flight instructions have completed. However, under rare circumstances, the architecturally-visible overlap of instructions might be observable.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T && MVE**.*

$I_{BPHG}$ If a higher priority exception preempts the generated watchpoint, then the in-flight instruction might be visible when the PE enters Debug state or a DebugMonitor exception is subsequently taken. The debugger and software can observe the stacked value of EPSR.ECI to determine the overlap status.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T && MVE && Halting debug || DebugMonitor exception**.*

$R_{HKDY}$ Watchpoints can be triggered from a number of comparators based on configurable events. If triggered from an instruction address comparator, the watchpoint is triggered whenever the instruction attempts to execute. Predication has no effect on the watchpoint address comparator.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T && MVE**.*

$R_{SJCQ}$ Predication affects data address and value comparators, and a watchpoint is triggered if the memory access is not predicated. It is not be triggered if the access is not performed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DWT-T && MVE**.*

$R_{ZHVG}$ If Data value matching is supported, then the DWT implements the DWT_VMASK<n> registers for each implemented comparator *n* that supports Data value matching.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

### B13.2.3 Constraints on programming DWT comparators

$R_{MSPS}$ If a DWT comparator, <n>, or pair of comparators, <n> and <n+1>, is programmed with a reserved combination of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION, then it is UNPREDICTABLE whether any comparator:

- Behaves as if disabled.
- Generates a match, setting DWT_FUNCTION.MATCHED bit to an UNKNOWN value, and any of the following:
  - Asserts **CMPMATCH**.
  - Generates a debug event.
  - Generates one or more trace packets.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GPLQ}$ Combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION that are not specified as valid combinations are reserved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{JHZK}$ It is IMPLEMENTATION DEFINED which values of DWT_FUNCTION.MATCH are valid for counter <n>. DWT_FUNCTION.ID defines which values are valid. Values that are not valid are reserved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CNHN}$ The valid combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION for a single comparator, and the events and Data trace packets that the comparator can generate from matching a single access, are identified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**Yes**: means that the packet or event is generated on a comparator match.

| Comparator Type | MATCH | ACTION | Debug Event | Data Trace: Match Packet | Data Trace: PC Value Match Packet | Data Trace: Data Address Packet | Data Trace: Data Value Packet |
|---|---|---|---|---|---|---|---|
| Disabled | 0b0000 | 0bxx | - | - | - | - | - |
| Cycle Counter | 0b0001 | 0b00 | - | - | - | - | - |
| | | 0b01 | Yes | - | - | - | - |
| | | 0b10 | - | Yes | - | - | - |
| | | 0b11 | - | - | Yes | - | - |
| Instruction Address | 0b0010 | 0b00 | - | - | - | - | - |
| | | 0b01 | Yes | - | - | - | - |
| | | 0b10 | - | Yes | - | - | - |
| Data address | 0b01xx (not 0b0111) | 0b00 | - | - | - | - | - |
| | | 0b01 | Yes | - | - | - | - |
| | | 0b10 | - | Yes | - | - | - |
| | | 0b11 | - | - | Yes | - | - |
| Data value | 0b10xx (not 0b1011) | 0b00 | - | - | - | - | - |
| | | 0b01 | Yes | - | - | - | - |
| | | 0b10 | - | Yes | - | - | - |
| Data address with value | 0b11xx (not 0b1111) | 0b10 | - | - | - | - | Yes |
| | | 0b11 | - | - | Yes | - | Yes |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**. Note, Cycle counter, Data value and Data address with value are only available if M is implemented.*

## Instruction address range

$R_{DKHG}$ To match an instruction that is in an instruction address range, the following conditions are met:

- The first comparator, *<n-1>*, is programmed for *Instruction address.*
- The second comparator, *<n>,* is programmed for *Instruction address limit.*

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{LNQD}$ The valid combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION for an instruction address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**First**: means that the packet or event is generated by the first comparator match.

**Second**: means that the packet or event is generated by the second comparator match.

| MATCH <n-1> | MATCH <n> | ACTION <n-1> | ACTION <n> | Data Trace: Debug Event | Data Trace: Match packet | Data Trace: PC Value packet | Data Trace: Data Address packet | Data Trace: Data Value packet |
|---|---|---|---|---|---|---|---|---|
| 0b0000 | 0b0011 | 0bxx | 0bxx | - | - | - | - | - |
| 0b0010 | 0b0011 | 0b00 | 0b00 | - | - | - | - | - |
| | | 0b00 | 0b11 | - | - | Second | - | - |
| | | 0b01 | 0b00 | First | - | - | - | - |
| | | 0b10 | 0b00 | - | First | - | - | - |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T** && **M**.*

$R_{VDRJ}$ If the Main Extension is not implemented the valid combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION for an instruction address range, and the events and data trace packets that matching

a single access can generate, are specified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**First**: means that the packet or event is generated by the first comparator match.

**Second**: means that the packet or event is generated by the second comparator match.

| MATCH | | ACTION | | Data Trace | | | | |
|---|---|---|---|---|---|---|---|---|
| <n-1> | <n> | <n-1> | <n> | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b0011 | 0bxx | 0bxx | - | - | - | - | - |
| 0b0010 | 0b0011 | 0b00 | 0b00 | - | - | - | - | - |
| | | 0b01 | 0b00 | First | - | - | - | - |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && !M**.*

## Data address range

$R_{LDGR}$ To match a data access in a data address range, the following conditions are met:

- The first comparator, <*n-1*>, is programmed for either *Data address* or *Data address with value*.
- The second comparator, <*n*>, is programmed for *Data address limit.*

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M**.*

$R_{PSBJ}$ The valid combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION for a data address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**First**: means that the packet or event is generated by the first comparator match.

**Second**: means that the packet or event is generated by the second comparator match.

| MATCH | | ACTION | | Data Trace | | | | |
|---|---|---|---|---|---|---|---|---|
| <n-1> | <n> | <n-1> | <n> | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b0111 | 0bxx | 0bxx | - | - | - | - | - |
| 0b01xx (not 0b0111) | 0b0111 | 0b00 | 0b00 | - | - | - | - | - |
| | | 0b00 | 0b11 | - | - | - | Second | - |
| | | 0b01 | 0b00 | First | - | - | - | - |
| | | 0b10 | 0b00 | - | First | - | - | - |
| | | 0b11 | 0b00 | - | - | First | - | - |
| | | 0b11 | 0b11 | - | - | First | Second | - |
| 0b11xx (not 0b1111) | 0b0111 | 0b10 | 0b00 | - | - | - | - | First |
| | | 0b10 | 0b11 | - | - | - | Second | First |
| | | 0b10 | 0b11 | - | - | First | - | First |
| | | 0b11 | 0b11 | - | - | First | Second | First |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{HDMX}$ If the Main Extension is not implemented the valid combinations of and for a data address range, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**First**: means that the packet or event is generated by the first comparator match.

**Second**: means that the packet or event is generated by the second comparator match.

| MATCH | | ACTION | | Data Trace | | | | |
|---|---|---|---|---|---|---|---|---|
| <n-1> | <n> | <n-1> | <n> | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b0111 | 0bxx | 0bxx | - | - | - | - | - |
| 0b01xx | 0b0111 | 0b00 | 0b00 | - | - | - | - | - |
| (not | | 0b00 | 0b11 | - | - | - | Second | - |
| 0b0111) | | 0b01 | 0b00 | First | - | - | - | - |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && !M**.*

## Data value at specific address

$R_{KFHV}$ Matching data values at specific data addresses is possible only if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NNXD}$ To match a data value at a specific data address, the following conditions are met:

- The first comparator, *<n-1>*, is programmed for either *Data address* or *Data address with value*.
- The second comparator, *<n>*, is programmed for *Linked data value.*

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{JKGJ}$ The first comparator matches any access that matches the address. The second matches only accesses that match the address and the data value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NTSD}$ The valid combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION for a linked data value, and the events and data trace packets that matching a single access can generate, are specified in the following table.

In the table:

**-**: means that the packet or event is not generated.

**First**: means that the packet or event is generated by the first comparator match.

**Second**: means that the packet or event is generated by the second comparator match.

**Both**: means that a first packet is generated by a first comparator match, even if the Linked data value comparator does not match, and a second packet is generated by the second comparator match, if both comparators match.

| MATCH | | ACTION | | Data Trace | | | | |
|---|---|---|---|---|---|---|---|---|
| <n-1> | <n> | <n-1> | <n> | Debug Event | Match packet | PC Value packet | Data Address packet | Data Value packet |
| 0b0000 | 0b1011 | 0bxx | 0bxx | - | - | - | - | - |
| 0b01xx (not 0b0111) | 0b1011 | 0b00 | 0b00 | - | - | - | - | - |
| | | 0b00 | 0b01 | Second | - | - | - | - |
| | | 0b00 | 0b10 | - | Second | - | - | - |
| | | 0b01 | 0b00 | First | - | - | - | - |
| | | 0b01 | 0b10 | First | Second | - | - | - |
| | | 0b10 | 0b00 | - | First | - | - | - |
| | | 0b10 | 0b01 | Second | First | - | - | - |
| | | 0b11 | 0b00 | - | - | First | - | - |
| | | 0b11 | 0b01 | Second | - | First | - | - |
| | | 0b11 | 0b10 | - | Second | First | - | - |
| 0b11xx (not 0b1111) | 0b1011 | 0b10 | 0b00 | - | - | - | - | First |
| | | 0b10 | 0b01 | Second | - | - | - | First |
| | | 0b10 | 0b10 | - | Second | - | - | First |
| | | 0b11 | 0b00 | - | - | First | - | First |
| | | 0b11 | 0b01 | Second | - | First | - | First |
| | | 0b11 | 0b10 | - | Second | First | - | First |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### B13.2.4 CMPMATCH trigger events

$I_{VNCC}$ The **CMPMATCH** events signal watchpoint matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{PRJG}$ The implementation of **CMPMATCH** is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FTWC}$ If an ETM is implemented, **CMPMATCH** events are output to the ETM.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && ETM**.*

$R_{TMZX}$ If an ETM is not implemented, the effect of **CMPMATCH** is IMPLEMENTATION DEFINED, including whether the trigger event has any observable effect or whether observable effects are visible to other components in the system.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{XXKM}$ For all enabled watchpoints, if DWT_FUNCTIONn is not programmed as an Instruction address limit comparator and is not programmed as a Data address limit comparator, **CMPMATCH[n]** is triggered on a comparator match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GVHS}$ For all enabled watchpoints, if DWT_FUNCTIONn is programmed as an Instruction address limit or Data address limit comparator, it is UNPREDICTABLE whether **CMPMATCH[n]** is triggered on a comparator match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### B13.2.5 Matching in detail

#### Instruction address matching in detail

$R_{GNVB}$ The DWT checks all instructions that are executed by a simple sequential execution of the program and do not generate any exception for an instruction address match, including conditional instructions that fail their condition code check.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NQGR}$ An instruction might be checked by the DWT for an instruction address match if it either:

- Is executed by a simple sequential execution of the program and generates a synchronous exception.
- Would be executed by the sequential execution of the program but is abandoned because of an asynchronous exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KJJC}$ Speculative instruction prefetches, other than those that would be executed by the sequential execution of the program but that are abandoned because of asynchronous exceptions, do not generate matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{DSDT}$ For all instruction address matches, if bit[0] of the comparator address has a value of 1, it is UNPREDICTABLE whether a match is generated when the other address bits match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KLXM}$ For single instruction address matches, an instruction matches if the address of the first byte of the instruction matches the comparator address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FXFM}$ For single address matches, if the instruction at address A is a 4-byte T32 instruction, and the address A+2 matches but the address A does not match, it is UNPREDICTABLE whether a match is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{DNKD}$ For instruction address range matches, an instruction at address A matches if the address A lies between the lower comparator address, which is specified by comparator <*n-1*>, and the limit comparator address, which is specified by comparator <*n*>. Both addresses are inclusive to the range.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{JNXZ}$ For instruction address range matches, if the instruction at address A is a 4-byte T32 instruction, and the address A+2 lies in the range but the address A does not lie in the range, it is UNPREDICTABLE whether a match is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{MLMQ}$ For instruction address range matches, if so configured, a Data trace PC value packet or Data trace match packet is generated for the first instruction that is executed in the range.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{VHHW}$ For instruction address range matches, if so configured, a branch or sequential execution that stays within the range does not necessarily generate a new packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{HMNX}$ For instruction address range matches, if so configured, **CMPMATCH**[n-1] is triggered for each instruction that is executed inside the range, where *n-1* is the lower of the two comparators that configure the range.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### Data address matching in detail

$R_{BPWC}$ For all Data Address matches, all bits of the comparator address are considered.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GSLX}$ Speculative reads might generate data address matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{WWBH}$ Speculative writes do not generate data address matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VJFB}$ Prefetches into a cache do not generate data address matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CMRP}$ For single data address matches, an access matches if any accessed byte lies between the comparator address and a limit that is defined by DWT_FUNCTION.DATAVSIZE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KHRF}$ For single data address matches, the comparator address is naturally aligned to DWT_FUNCTION.DATAVSIZE otherwise generation of watchpoint events is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KKRJ}$ For data address range matches, an access matches if any accessed byte lies between the lower comparator address, which is specified by comparator <*n-1*>, and the limit comparator address, which is specified by comparator <*n*>. Both addresses are inclusive to the range.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CFMR}$ For data address range matches, DWT_FUNCTION.DATAVSIZE is set to `0b00` for both the lower comparator address and the limit comparator address otherwise it is UNPREDICTABLE whether or not a match is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### Data value matching in detail

$R_{BMSM}$ Data value matching is only possible if the Main Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FVFQ}$ Speculative reads might generate data value matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VGJF}$ Speculative writes do not generate data value matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{MLFK}$ Prefetches into a cache do not generate data value matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{RMDB}$ For data value matches, if the access size is smaller than DWT_FUNCTION.DATAVSIZE, there is no match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{ZDPM}$ For unlinked data value matches, an access matches if all bytes of any naturally-aligned subset, the size of which is specified by DWT_FUNCTION.DATAVSIZE, of the access match the data value in DWT_COMPn.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{ZHXP}$ The data value in DWT_COMPn is in little-endian order with respect to memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{HMMS}$ If the access is unaligned then this might generate a higher priority alignment fault, depending on the instruction type, profile, and configuration. In these cases no match is generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{SQKS}$ For unlinked data value matches, if an access is unaligned, it is IMPLEMENTATION DEFINED whether it either treated as:

- A sequence of byte accesses.
- A sequence of naturally-aligned accesses covering the accessed bytes. For a read, this access might access more bytes than the original access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QRPW}$ For linked data value matching, if an access is larger than DWT_FUNCTION.DATAVSIZE, then only the naturally-aligned subset of the access of size DWT_FUNCTION.DATAVSIZE at the matching address is compared for a match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QVRK}$ For linked data value matching, the data address comparator address is naturally aligned to

DWT_FUNCTION.DATAVSIZE, and the DWT_FUNCTION.DATAVSIZE values for both comparators are the same.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KRCV}$ A Data value comparator that is linked to a Data address comparator does not change the behavior of the address comparator.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KQJB}$ For each comparator *n* that is configured to Data Value or Linked Data Value matching it is UNPREDICTABLE whether comparator *n* generates a match when for bit *m*=31-0, if any of the following are true:

- DWT_FUNCTION<n>.DATAVSIZE specifies halfword or byte comparison and DWT_COMPn[31:16] is not equal to DWT_COMPn[15:0].
- DWT_FUNCTION<n>.DATAVSIZE specifies byte comparsion and DWT_COMPn[15:8] is not equal to DWT_COMPn[7:0].

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{ZKRZ}$ For each comparator *n* that is configured to Data Value or Linked Data Value matching the value matches if, for each bit[*m*] any of the following are true:

- Bit[*m*] of the value is equal to DWT_COMPn[*m*].
- DWT_VMASKn is implemented, DWT_COMPn[*m*] is set to 0, and DWT_VMASKn[*m*] is set to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

$R_{BVLK}$ For each comparator *n* that is configured for Data Value or Linked Value matching, if DWT_VMASKn is implemented, then it is UNPREDICTABLE whether comparator *n* generates a match when, for bit *m* = 0-31, all of the following are true:

- DWT_VMASKn[*m*] is set to 1.
- DWT_COMPn[*m*] is not set to 0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

$R_{HRTJ}$ For each comparator *n* that is configured for Data Value or Linked Value matching, if DWT_VMASKn is implemented, then it is UNPREDICTABLE whether comparator *n* generates a match if any of the following are true:

- DWT_FUNCTION<n>.DATAVSIZE specifies halfword or byte comparison and DWT_VMASKn[31:16] is not equal to DWT_VMASKn[15:0].
- DWT_FUNCTION<n>.DATAVSIZE specifies byte comparison and DWT_VMASKn[15:8] is not equal to DWT_VMASKn[7:0].

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

$R_{SMHR}$ For each comparator *n* that is configured for neither Data Value nor Linked Data Value matching, DWT_VMASKn is ignored if it is implemented.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **DSPDE**.*

See also:

`DWT_AddressCompare()`.

`DWT_ValidMatch()`.

`DWT_InstructionAddressMatch()`.

`DWT_DataAddressMatch()`.

`DWT_DataValueMatch()`.

### B13.2.6 DWT match restrictions and relaxations

$R_{FRWG}$ It is IMPLEMENTATION DEFINED whether the DWT treats a fetch from the exception vector table as part of an exception entry or reset as a data access or ignores these accesses, for the purposes of DWT comparator matches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{DTHW}$ A fetch by the DWT from the exception vector table as part of an exception entry is never treated as an instruction fetch.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{JQHW}$ If a return is tail-chained, it is IMPLEMENTATION DEFINED whether hardware accesses the stack and therefore IMPLEMENTATION DEFINED whether the DWT can generate events or trace.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VJTK}$ The DWT does not match accesses from the DAP.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{MNBX}$ Any executed `NOP` or `IT` that matches an appropriately configured instruction address watchpoint causes a match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{SLPX}$ It is IMPLEMENTATION DEFINED whether a failed `STREX` instruction can generate a data access match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NHLN}$ If an instruction or operation makes multiple or unaligned data accesses, then it is UNPREDICTABLE whether any nonmatching access generated by an instruction that generated a matching access is treated as a matching access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CSSQ}$ If an instruction or operation makes multiple or unaligned data accesses, then CMPMATCH is triggered for each matching access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VFXT}$ If an instruction or operation makes multiple or unaligned data accesses, then, if so configured, only a data value match of at least a part of the value that is guaranteed to be single-copy atomic can generate a match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{WJNR}$ If an instruction or operation makes multiple or unaligned data accesses, then, if so configured, for a matching data access that generates a debug event, if permitted, DHCSR.C_HALT or DEMCR.MON_PEND, as applicable, is set to 1.

A pending DebugMonitor exception does not interrupt the multiple accesses, but another interrupt might, which means that the debug event might be taken before the multiple operations complete.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QCJL}$ The DWT can match on the address of an access that generates a BusFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QVHL}$ It is IMPLEMENTATION DEFINED whether a stored value for an access that generates a BusFault:

- Can generate a data value match.
- Can be traced.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KLFC}$ For a load access that returns a BusFault, any data that is returned by the memory system is invalid, and the DWT does not:

- Generate a data value match.
- Generate a Data trace data value packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{TQCF}$ A data access that generates any fault other than a BusFault does not generate a data address or data value match at the DWT and is not traced.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{FRHP}$ DWT matches are generated asynchronously.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{THHR}$ A DSB barrier guarantees that the effect of a DWT match is visible to a subsequent read of DWT_FUNCTION.-MATCHED, DHCSR, or DEMCR. In the absence of a DSB barrier, the effect is only guaranteed to be visible in finite time.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{HPGH}$ The effects of a DWT match never affect instructions appearing in program order before the operation that generates the match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

See also:

B3.26 *Tail-chaining* on page 123.

### B13.2.7 DWT trace restrictions and relaxations

$R_{HDKK}$ Where a single instruction or operation, or multiple instructions, generate multiple accesses that each generate one or more trace packets, then if the architecture guarantees the order in which a pair of these accesses is observed by the PE, the first trace packets that are generated for each of those accesses appear in the trace output in the same order.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{WSKK}$ Where a single instruction or operation, or multiple instructions, generate multiple accesses that each generate one or more trace packets, then if the architecture does not guarantee the order of the accesses, the order of the trace packets in the trace output is not defined.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{XCNB}$ If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, only the first access is guaranteed to generate a Data trace PC value packet, Data trace data address packet, or Data trace match packet. If the architecture does not guarantee the order of the accesses, the first access might be any of the accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{XVBT}$ If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, a Data trace data value packet is generated for each matching access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QSCF}$ If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, it is UNPREDICTABLE how many Data trace data value packets are generated for each unaligned matching access. An implementation might over-read, meaning that more data outside the access might be traced.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KXBL}$ If a single instruction or operation makes multiple or unaligned data accesses, then, if so configured, for a matching data access that generates a Data trace data value packet, at least that part of the value that is guaranteed to be single-copy atomic is traced.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QWQS}$ Duplicate Data trace PC value packets, Data trace data address packets, and Data trace data value packets from a single access are not generated for a single access.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{CPXW}$ Where a comparator or linked pair of comparators generates multiple packet types for a single access, the packets appear in the trace output in the following order:

1. Data trace PC value packet.
2. Data trace match packet, generated by a Data address or Data address with value comparator match.
3. Data trace data address packet.
4. Data trace match packet, generated by a Data value comparator match.
5. Data trace data value packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{QXBC}$ Where a comparator or linked pair of comparators generates multiple packet types for a single access, packets are not interleaved with packets that are generated by other accesses by the same comparator or linked pair of comparators.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{RHNF}$ Where a comparator or linked pair of comparators generates a trace packet for a single access, if a comparator other than this comparator or this linked pair of comparators generates a trace packet of the same type for the same access, then only one of these packets is output. It is IMPLEMENTATION DEFINED which comparator is chosen.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{MJXG}$ Arm recommends that the packet from the lowest-numbered comparator is output.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{DKMV}$ Where a comparator or linked pair of comparators generates multiple packet types for a single access, if any of the packets cannot be output and an Overflow packet is generated, then the remaining packets for that access are not generated.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{LNBW}$ Where a comparator or linked pair of comparators generates multiple packet types for a single access, packets might be interleaved with packets that are generated for the same access by comparators other than this comparator or this linked pair of comparators.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### B13.2.8 CYCCNT cycle counter and related timers

$R_{SVPW}$ CYCCNT is an optional free-running 32-bit cycle counter. If the DWT unit implements CYCCNT then DWT_CTRL.NOCYCCNT is RAZ.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KRFP}$ When implemented and enabled, CYCCNT increments on each cycle of the PE clock.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NFJW}$ When the counter overflows it transparently wraps to zero.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GXJK}$ DWT_CTRL.CYCCNTENA enables the CYCCNT counter.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{BKCG}$ POSTCNT is a 4-bit countdown counter derived from CYCCNT, that acts as a timer for the periodic generation of Periodic PC sample packets or Event counter packets, when these packets are enabled.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{MGGL}$ Periodic PC sample packets are not the same as the Data trace PC value packets that are generated by the DWT comparators.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{DKGR}$ The DWT does not support the generation of Periodic PC sample packets or Event packets if it does not implement the CYCCNT timer and DWT_CTRL.NOTRCPKT is RAO.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{RNTV}$ The DWT_CTRL.CYCTAP bit selects the CYCCNT tap bit for POSTCNT.

| CYCTAP bit | CYCCNT tap at | POSTCNT clock rate |
|---|---|---|
| 0 | Bit[6] | (PE clock)/64 |
| 1 | Bit[10] | (PE clock)/1024 |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{SXKK}$ A write to DWT_CTRL will initialize POSTCNT to the previous value of DWT_CTRL.POSTINIT if all of the following are true:

- DWT_CTRL.PCSAMPLENA was set to 0 prior to the write.
- DWT_CTRL.CYCEVTENA was set to 0 prior to the write.
- The write sets either DWT_CTRL.PCSAMPLENA or DWT_CTRL.CYCEVTENA to 1.

It is UNPREDICTABLE whether any other write to DWT_CTRL that alters the value of DWT_CTRL.PCSAMPLENA and DWT_CTRL.CYCEVTENA sets POSTCNT to DWT_CTRL.POSTINIT or leaves POSTCNT unchanged.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{XFRM}$ When either DWT_CTRL.PCSAMPLENA or DWT_CTRL.CYCEVTENA is set to 1, and the CYCCNT tap bit transitions, either from 0 to 1 or from 1 to 0:

- If POSTCNT is nonzero, POSTCNT decrements by 1.
- If POSTCNT is 0, the DWT:
  - Reloads POSTCNT from DWT_CTRL.POSTPRESET.
  - Generates a Periodic PC Sample packets if DWT_CTRL.PCSAMPLENA is set to 1.
  - Generates an Event Counter packet with the Cyc bit set to 1 if DWT_CTRL.CYCEVTENA is set to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{PNNS}$ The enable bit for the POSTCNT counter underflow event is DWT_CTRL.CYCEVTENA. There is no overflow event for the CYCCNT counter. When CYCCNT overflows it wraps to zero transparently. Software cannot access the POSTCNT value directly, or change this value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{JRVV}$ This means that, to initialize POSTCNT, software:

1. Ensures that DWT_CTRL.CYCEVTENA and DWT_CTRL.PCSAMPLENA are set to 0. This can be achieved with a single write to DWT_CTRL. This is also the reset value of these bits.
2. Writes the required initial value of POSTCNT to the DWT_CTRL.POSTINIT field, leaving DWT_CTRL.CYCEVTENA and DWT_CTRL.PCSAMPLENA set to 0.
3. Sets either DWT_CTRL.CYCEVTENA or DWT_CTRL.PCSAMPLENA to 1 to enable the POSTCNT counter.

Each of these are separate writes to DWT_CTRL.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{KNHF}$ Disabling CYCCNT stops POSTCNT.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{TMHN}$ Writes to DWT_CTRL.POSTINIT are ignored if either DWT_CTRL.CYCEVTENA was set to 1 or DWT_CTRL.PCSAMPLENA was set to 1 prior to the write.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

### B13.2.9 Profiling counter support

$I_{HXPV}$ If the Main Extension is implemented profiling counter support is an optional Non-invasive debug feature.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M**.*

$R_{WHWR}$ If profiling counter support is implemented the DWT provides five 8-bit Event counters for software profiling:

- DWT_FOLDCNT.
- DWT_LSUNCT.
- DWT_EXCCNT.
- DWT_SLEEPCNT.
- DWT_CPICNT.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{GLMJ}$ Event counters do not increment when the PE is halted.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{BRGW}$ The Event counters provide broadly accurate and statistically useful count information. However, the architecture allows for a reasonable degree of inaccuracy in the counts.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{WMNV}$ The Event counters use the same definition of cycle in particular when counting cycles in power-saving modes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T && M**.*

$I_{GNWQ}$ To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- Under normal operating conditions, the Event counters present an accurate value count.
- Entry to or exit from Debug state can be a source of inaccuracy.
- Under very unusual, non-repeating pathological cases, the counts can be inaccurate.

An implementation does not introduce inaccuracies that can be triggered systematically by the execution of normal pieces of software. As the Event counters include counters for measuring exception overhead, this includes the operation of exceptions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{CHKR}$ Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the Event counters are expected.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{MWGQ}$ At entry and exit from an exception or sleep state, the exact attribution of cycles to the exception and cycles to the sleep overhead counters is IMPLEMENTATION DEFINED. Arm recommends that the overhead cycles are attributed to the overhead counters.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{MPQN}$ The architecture does not define the point in a pipeline where the particular instruction increments an Event counter, relative to the point where the incremented counter can be read.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{LMPK}$ An Event counter overflows on every 256th event that is counted and then wraps to 0. If the appropriate counter overflow event is enabled in DWT_CTRL the DWT outputs an Event counter packet with the appropriate counter flag set to 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{LHMB}$ Setting one of the enable bits to 1 clears the corresponding counter to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{QRPG}$ The following equation holds:

$$ICNT = CNT_{CYCLES} + CNT_{FOLD} - (CNT_{LSU} + CNT_{EXC} + CNT_{SLEEP} + CNT_{CPI})$$

Where:

$ICNT$: is the total number of instructions Architecturally executed.

$CNT_{CYCLES}$: is the number of cycles counted by DWT_CYCCNT.

$CNT_{FOLD}$: is the number of instructions counted by DWT_FOLDCNT.

$CNT_{LSU}$: is the number of cycles counted by DWT_LSUNCT.

$CNT_{EXC}$: is the number of cycles counted by DWT_EXCCNT.

$CNT_{SLEEP}$: is the number of cycles counted by DWT_SLEEPCNT.

$CNT_{CPI}$: is the number of cycles counted by DWT_CPICNT.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

See also:

B13.4 *Trace Port Interface Unit* on page 357.

### Generating Overflow packets from Event counters

$R_{KWDH}$ If an Event counter wraps to zero and the previous Event counter packet has been delayed and has not yet been output, and the counter flag in the previous Event counter packet is set to 0, then it is IMPLEMENTATION DEFINED whether:

- The DWT attempts to generate a second Event counter packet.
- The DWT updates the delayed Event counter packet to include the new wrap event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{HKTL}$ If an Event counter wraps to zero and the previous Event counter packet has been delayed and has not yet been output, and the counter flag in the previous Event counter packet is set to 1, the DWT attempts to generate a second Event counter packet.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{VPXK}$ If the DWT unit attempts to generate a packet when its output buffer is full, an Overflow packet is output.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{SFFL}$ The size of the DWT output buffer is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

## B13.2.10 Program Counter sampling support

$R_{FXWL}$ Program Counter sampling is an optional component provided through DWT_PCSR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{LNJL}$ Program Counter sampling is independent of PC sampling provided by:

- Periodic PC sample packets.

- Data trace PC value packets generated as a result of a DWT comparator match.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{KVFB}$ The architecture does not define the delay between an instruction being executed by the PE and its address being written to DWT_PCSR.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NGNT}$ When DWT_PCSR returns a value other than `0xFFFFFFFF`, the returned value is an instruction that has been committed for execution. It is IMPLEMENTATION DEFINED whether an instruction that failed its condition code check is considered as committed for execution. A read of DWT_PCSR does not return the address of an instruction that has been fetched but not committed for execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{KCBH}$ Arm recommends that instructions that fail the condition code check are considered as committed instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{WPMF}$ DWT_PCSR is able to sample references to branch targets. It is IMPLEMENTATION DEFINED whether it can sample references to other instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{SJVK}$ Arm recommends that DWT_PCSR can sample a reference to any instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{LMDG}$ The branch target for a conditional branch that fails its Condition code check is the instruction that immediately follows the conditional branch instruction. The branch target for an exception is the exception vector address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{NWKP}$ Periodic sampling of DWT_PCSR provides broadly accurate and statistically useful profile information. However, the architecture allows for a reasonable degree of inaccuracy in the sampled data.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{TJTS}$ To keep the implementation and validation cost low, a reasonable degree of inaccuracy in the counts is acceptable. Arm does not define *a reasonable degree of inaccuracy* but recommends the following guidelines:

- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the sample to represent an instruction that was not committed for execution.
- Under unusual non-repeating pathological cases, the sample can represent an instruction that was not committed for execution. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in sampling is very unlikely.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{KVJM}$ Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the sampled data are expected.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$R_{JMVS}$ When DEMCR.TRCENA is set to 0 any read of DWT_PCSR returns an UNKNOWN value.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**.*

$I_{PXMR}$ A read of DWT_PCSR will return `0xFFFFFFFF` if any of the following are true:

- The PE is in Debug state.
- The instruction was executed in Secure state and `SecureNoninvasiveDebugAllowed()` returns FALSE.
- `NoninvasiveDebugAllowed()` returns FALSE.
- The address of a recently executed instruction is not available.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **DWT-T**. Note, S is requried for Secure state.*

## B13.3 Embedded Trace Macrocell

$I_{LCCX}$ An *Embedded Trace Macrocell* (ETM) is an optional non-invasive debug feature of an Armv8-M implementation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{NGTT}$ An ETM implementation complies with one of the following versions of the ETM architecture:

| Data trace | Security Extension | |
|---|---|---|
| | **Implemented** | **Not implemented** |
| Implemented | ETMv3 not permitted | ETMv3 not permitted |
| | ETMv4, version 4.2 or later | ETMv4, version 4.0 or later |
| Not Implemented | ETMv3, version 3.5 or later | ETMv3, version 3.5 |
| | ETMv4, version 4.2 or later | ETMv4, version 4.0 or later |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{QWRD}$ If Armv8.1-M is implemented and an ETM is implemented, ETMv4 version 4.5 is required. Data trace is not supported in ETMv4 version 4.5 and is not supported in Armv8.1-M.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **ETM**.*

$R_{LPJM}$ If an ETM is implemented a trace sink is also implemented. If the trace sink that is implemented is the TPIU it is CoreSight compliant, and complies with the TPIU architecture for compatibility with Arm and other CoreSight-compatible debug solutions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{NLNS}$ When an Armv8-M implementation includes an ETM, the **CMPMATCH[N]** signals from the DWT unit are available as control inputs to the ETM unit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{NJDK}$ If the Main Extension is not implemented, it is IMPLEMENTATION DEFINED whether the ETM is accessible only to the debugger and is RES0 to software.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM && !M**.*

$R_{WPBN}$ If the ETMv3 is implemented the debugger programs the ETMTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{TJSF}$ If the ETMv4 is implemented the debugger programs the TRCTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

$R_{WSTB}$ The ETM is not directly affected by DEMCR.TRCENA being set to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **ETM**.*

See also:

*Arm® CoreSight™ Architecture Specification*.

B13.2.4 *CMPMATCH trigger events* on page 343.

## B13.4 Trace Port Interface Unit

$I_{PWXP}$ The *Trace Port Interface Unit* (TPIU) support for Armv8-M provides an output path for trace data from the DWT, ITM, and ETM. The TPIU is a trace sink.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{CRTQ}$ It is IMPLEMENTATION DEFINED whether the TPIU supports a parallel trace port output.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{GTRP}$ It is IMPLEMENTATION DEFINED whether the TPIU supports low-speed asynchronous serial port output using NRZ encoding. This operates as a traditional UART.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{LKQT}$ It is IMPLEMENTATION DEFINED whether the TPIU supports medium-speed asynchronous serial port output using Manchester encoding.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$I_{SDDK}$ Arm recommends that the TPIU provides both parallel and asynchronous serial ports, for maximum flexibility with external capture devices.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{HJXK}$ Whether the trace port clock is synchronous to the PE clock is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{PKKS}$ It is IMPLEMENTATION DEFINED whether the TPIU is reset by a Cold reset or has an independent Cold reset.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{JBKJ}$ Software ensures that all trace is output and flushed to the trace sink before setting the DEMCR.TRCENA bit to 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{STLV}$ The TPIU is not directly affected by DEMCR.TRCENA being set to 0 or `NoninvasiveDebugAllowed()` being FALSE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{JLCQ}$ The output formatting modes that are supported by the TPIU are IMPLEMENTATION DEFINED. They are:

- Bypass.
- Continuous.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{DMFP}$ Bypass mode is only supported if a serial port output is supported.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{RRJP}$ Continuous mode is supported if the parallel trace port is implemented. Continuous mode is selected when the parallel trace port is used.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

$R_{FCFT}$ Continuous mode is supported if the ETM is implemented. Continuous mode is selected when the ETM is used.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **TPIU**.*

See also:

*TPIU_FFCR, Formatter and Flush Control Register*.

B13.1 *Instrumentation Trace Macrocell* on page 325.

B13.3 *Embedded Trace Macrocell* on page 356.

Chapter B1 *Resets* on page 60.

# B13.5 Flash Patch and Breakpoint unit

## B13.5.1 About the FPB unit

$R_{FTWL}$ The *Flash Patch and Breakpoint* (FPB) unit supports setting breakpoints on instruction fetches.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$I_{BPFS}$ The name Flash Patch and Breakpoint unit is historical and the architecture does not support remapping functionality.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{GDWW}$ The number of implemented instruction address comparators is IMPLEMENTATION DEFINED. Software can discover the number of implemented instruction address comparators from FP_CTRL.NUM_CODE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

See also:

Chapter B7 *The System Address Map* on page 240.

B13.2.7 *DWT trace restrictions and relaxations* on page 349.

Chapter D1 *Register Specification* on page 1303.

## B13.5.2 FPB unit operation

$R_{RKFD}$ The FPB contains the following register types:

- A general control register, FP_CTRL.
- Comparator registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{BKKW}$ Each implemented instruction address comparator supports breakpoint generation.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{FNQF}$ The FP_CTRL register provides a global enable bit for the FPB, and ID fields that indicate the numbers of instruction address comparison and literal comparison registers implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{CKBL}$ When configured for breakpoint generation, instruction address comparators can be configured to match any halfword-aligned addresses in the whole address map.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{XPXS}$ Instruction address comparators match only on instruction fetches. The FPB treats hardware accesses to the stack as data accesses for registers that are:

- Pushed to the stack by hardware as part of an exception entry or lazy state preservation.
- Popped from the stack by hardware as part of an exception return.
- Pushed to the stack by hardware as part of a Non-secure function return.
- Popped from the stack by hardware as part of a Non-secure function call.

It is IMPLEMENTATION DEFINED whether the FPB treats a fetch from the exception vector table as part of an exception entry as a data access, or ignores these accesses, for the purposes of FPB address comparator matches. The fetch is never be treated as an instruction fetch.

The FPB does not match accesses from the DAP.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$I_{CNBW}$ Bit[0] of each instruction fetch address is always 0.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{CJKK}$ When an Instruction address matching comparator is configured for breakpoint generation, a match on the address of a 32-bit instruction is configured to match the first halfword or both halfwords of the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{WSXN}$ If a Breakpoint debug event is generated by the FPB on the second halfword of a 32-bit T32 instruction, it is UNPREDICTABLE whether the breakpoint generates a debug event.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{XKJW}$ An FPB match specifying a Breakpoint debug event generates a Breakpoint debug event that halts the PE if all of the following conditions are true:

- `HaltingDebugAllowed()` == TRUE.
- DHCSR.C_DEBUGEN == 1.
- DHCSR.S_HALT == 0.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or DHCSR.S_SDE == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{JNVD}$ If the UDE is implemented an FPB match specifying a Breakpoint debug event generates a Breakpoint debug event that halts the PE in unprivileged mode if either of the following conditions are true:

- `HaltingDebugAllowed()` == TRUE
- `UnprivHaltingDebugAllowed()` == TRUE

and the all of the following conditions are true:

- DAUTHCTRL.UIDEN == 1.
- DHCSR.S_HALT == 0.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or DHCSR.S_SDE == 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FPB** && **UDE**.*

$R_{HXMP}$ An FPB match specifying a Breakpoint debug event generates a DebugMonitor exception if it does not halt the PE and all of the following conditions are true:

- DEMCR.MON_EN == 1.
- DHCSR.S_HALT == 0.
- The DebugMonitor exception group priority is greater than the current execution priority.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or DEMCR.SDME == 1.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{LTML}$ An FPB match specifying a Breakpoint debug event generates a DebugMonitor exception if it does not halt the PE and either of the following conditions are true:

- DEMCR.MON_EN == 1.
- DEMCR.UMON_EN == 1 and the PE is executing in unprivileged mode.

and all of the following conditions apply:

- DHCSR.S_HALT == 0.
- The DebugMonitor exception group priority is greater than the current execution priority.
- The Security Extension is not implemented, the matching instruction is executed in Non-secure state, or DEMCR.SDME == 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FPB && UDE**.*

$R_{BFPK}$ An FPB match that specifies a Breakpoint debug event is ignored if it does not meet the conditions for generating either:

- A Breakpoint debug event that halts the PE.
- A DebugMonitor exception.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{CLNV}$ Between a change to the debug authentication interface, DHCSR or DEMCR, that disables debug, and a following context synchronization event, it is UNPREDICTABLE whether any breakpoints generated by the FPB:

- Generate a Breakpoint debug event based on the old values and either:
  - If the Main Extension is implemented, generate a DebugMonitor exception.
  - Halts the PE.
- Are ignored.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

$R_{TKNR}$ Between a change to the debug authentication interface, DHCSR, DEMCR or DAUTHCTRL, that disables debug, and a following context synchronization event, it is UNPREDICTABLE whether any breakpoints generated by the FPB:

- Generate a Breakpoint debug event based on the old values and either:
  - If the Main Extension is implemented, generate a DebugMonitor exception.
  - Halts the PE.
- Are ignored.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FPB && UDE**.*

$R_{DZWB}$ Breakpoints that occur as a result of FPB Unit events behave in the same way as a scalar BKPT instruction. All in-flight instructions are completed before halting.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FPB && MVE**.*

$I_{YVFX}$ Entry to debug state and debug monitor is delayed until all in-flight instructions have completed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **FPB && MVE**.*

See also:

Halting debug.

B12.4.1 *About debug events* on page 302.

`GenerateDebugEventResponse()`

`InstructionExecute()`

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

`BKPTInstrDebugEvent()`

`FPB_BreakpointMatch()`

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### B13.5.3 Cache maintenance

$R_{BWSW}$ Instruction caches are not permitted to cache breakpoints that are generated by a Flash Patch and Breakpoint unit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **FPB**.*

# Chapter B14
# The Performance Monitoring Unit Extension

This chapter specifies the optional Armv8.1-M Performance Monitoring Unit (PMU) Extension. It contains the following sections:



*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

## B14.1 Counters

$I_{WBLR}$ The Performance Monitoring Unit (PMU) is an optional non-invasive debug component that allows events to be identified and counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{XVBC}$ There is space for a maximum of 31 IMPLEMENTATION DEFINED event counters in the PMU.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{DQQM}$ Each counter is either a 16-bit general-purpose event counter or a 32-bit cycle counter.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$I_{NNBX}$ By chaining counters in pairs, the counter range can be increased by halving the number of counters.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{VVVF}$ If the PMU is implemented, a minimum of two 16-bit event counters are required, and one 32-bit cycle counter.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$I_{JFVV}$ Each event counter can be configured to count any of the events that might be supported by an implementation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{SLDG}$ It is IMPLEMENTATION DEFINED whether PMU_EVCNTR*n* supports generating an interrupt. If the interrupt is not supported PMU_INTENSET.Pn is RAZ/WI.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{MGPL}$ Each event counter can be configured to increment on each occurrence of a specified performance event.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{VXPJ}$ The cycle counter is hard-wired to count cycles.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$R_{YVZG}$ In case of a counter chain event, the architecture guarantees that the unsigned overflow of the lower half of the counter and subsequent increment of the upper half of the counter are counted within the same cycle.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{RVJT}$ If a counter is configured to an event that is not supported on a specific implementation, the counter value is UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

## B14.2 Accuracy of the performance counters

$I_{RYDQ}$ The Performance Monitors provide broadly accurate and statistically useful count information. To keep the implementation and validation costs low, a reasonable degree of inaccuracy in the counts is acceptable.
Arm does not define a reasonable degree of inaccuracy, but recommends the following guidelines:

- Under normal operating conditions, the counters present an accurate value of the count.
- In exceptional circumstances, such as a change in Security state or other boundary condition, it is acceptable for the count to be inaccurate.
- Under very unusual, non-repeating, pathological cases, the counts can be inaccurate. These cases are likely to occur as a result of asynchronous exceptions, such as interrupts, where the chance of a systematic error in the count is very unlikely.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***PMU****.*

$R_{XCDG}$ A reasonable degree of inaccuracy in the PMU is permitted, if this does not create systematic inaccuracies in normal operating conditions.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***PMU****.*

## B14.3 Security, access, and modes

$I_{NPZL}$ The access to the PMU using the DAP mirrors that of the DWT.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{QDQC}$ The counters do not increment when:

- The PE is in Secure state and `SecureNoninvasiveDebugAllowed()` == FALSE.
- The PE is in Non-secure state and `NoninvasiveDebugAllowed()` == FALSE.
- The PE is in Debug state.
- The PE is in low-power state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**. Note, Secure state requires S.*

$R_{CMTX}$ When in low-power state or Debug state the counters retain their previous value.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{NNSQ}$ In lockup, it is UNKNOWN whether the counters increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{ZHBY}$ If PMU_CTRL.DP is RAZ, the PMU cycle counter increments regardless of the Security state of the PE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **S**.*

See also:

B13.2 *Data Watchpoint and Trace unit* on page 334.

B12.5 *Debug state* on page 318.

## B14.4 Attributability

$R_{GBQN}$ An event that is caused by the PE that is counting the event is Attributable. If an agent other than the PE that is counting the events causes an event, these events are Unattributable.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{XXLL}$ All architecturally defined events are Attributable, unless stated otherwise in the PMU event list.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{XGRG}$ Events caused by the execution of an instruction are Attributable to the Security state the instruction was executed in.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **S**.*

$R_{LDGQ}$ Events which are Attributable to Secure state are not counted if `SecureNoninvasiveDebugAllowed()` == FALSE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{WJDJ}$ Attributable Events caused by the following are Attributable to the NS-Req for the access:

- Vector fetches.
- The hardware stacking or unstacking of registers on any of:
  - Exception entry.
  - Exception exit.
  - Function call entry.
  - Function return.
  - Lazy state preservation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**. Note, Secure state requires S.*

$R_{VHLM}$ For each Unattributable event it is IMPLEMENTATION DEFINED whether the Unattributable event is counted when counting Attributable events is prohibited.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

See also:

B14.9 *Common event descriptions* on page 379.

## B14.5 Coexistence with the DWT Performance Monitors

$R_{HZKN}$ The PMU cycle counter PMU_CCNTR is an alias of the DWT_CYCCNT register. All derived functions are available whenever either the DWT or the PMU enables the cycle counter. If both the PMU and the DWT are implemented, DWT_CTRL.NOCYCCNT is RAZ.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-D**.*

$R_{ZBWD}$ When PMU_CTRL.E == 1:

- A read of DWT_CPICNT, DWT_EXCCNT, DWT_FOLDCNT, DWT_LSUCNT, and DWT_SLEEPCNT return an UNKNOWN value.
- DWT_CPICNT, DWT_EXCCNT, DWT_FOLDCNT, DWT_LSUCNT, and DWT_SLEEPCNT might increment at random.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-T**.*

$R_{KDKT}$ When any of DWT_CTRL.FOLDEVTENA, LSUEVTENA, SLEEPEVTENA, EXCEVTENA, CPIEVTENA == 1:

- A read of any PMU_EVCNTRn counters returns an UNKNOWN value.
- A read of the PMU overflow flags in PMU_OVSSET and PMU_OVSCLR return UNKNOWN values.
- The PMU_EVCNTRn counters might increment at random.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-T**.*

$R_{NDFK}$ When PMU_CTRL.E == 1 and any of DWT_CTRL.CYCEVTENA, FOLDEVTENA, LSUEVTENA, SLEEPEVTENA, EXCEVTENA, CPIEVTENA == 1:

- The generation of Event Counter packets by the DWT is UNPREDICTABLE.
- If any of the PMU_INTENSET.Pn flags are set to 1, the generation of interrupts is UNPREDICTABLE.
- If PMU_CTRL.TRO == 1 the generation of Event Counter packets by the PMU is UNPREDICTABLE.
- The PMU and DWT counters might increment at random.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-T**.*

$R_{RGQN}$ At any time:

- A write to any of DWT_CPICNT, DWT_EXCCNT, DWT_FOLDCNT, DWT_LSUCNT, or DWT_SLEEPCNT, including the indirect write of 0 when writing 1 to any of DWT_CTRL.CYCEVTENA, FOLDEVTENA, LSUEVTENA, SLEEPEVTENA, EXCEVTENA, CPIEVTENA, sets the PMU_EVCNTRn to UNKNOWN values.
- A write of PMU_EVCNTRn, including the indirect write of 0 when writing 1 to PMU_CTRL.P to 1, sets DWT_CPICNT, DWT_EXCCNT, DWT_FOLDCNT, DWT_LSUCNT, and DWT_SLEEPCNT to UNKNOWN values.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-T**.*

$I_{MZFG}$ It is permissible for PMU and DWT Counters to be aliased.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU && DWT-T**.*

## B14.6 Interrupts and Debug events

$I_{RJNK}$ Counters can be configured to generate interrupts or debug events on overflow.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{ZDMH}$ If a counter is configured to generate an interrupt when it overflows, DEMCR.MON_PEND is set to 1 to pend a DebugMonitor exception with DFSR.PMU set to 1. The associated field in PMU_OVSSET or PMU_OVSCLR indicates which counter triggered the exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{QZNR}$ If the corresponding bit in PMU_INTENSET or PMU_INTENCLR of the counter is set, and DEMCR.MON_EN is 1 then the counter is configured to generate an interrupt or debug event.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{WGGD}$ For each implemented event counter *m*, and the cycle counter, unsigned overflow of the counter halts the PE if all of the following conditions apply:

- PMU_INTENSET*m* is set to 1 for the event counter, or PMU_INTENSET.C is set to 1 for the cycle counter.
- `CanHaltOnEvent()` returns TRUE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **DSPDE**.*

$R_{YKXR}$ For each implemented event counter *m*, and the cycle counter, unsigned overflow of the counter pends a DebugMonitor exception if it does not halt the PE and all of the following conditions apply:

- PMU_INTENSET*m* is set to 1 for the event counter, or PMU_INTENSET.C is set to 1 for the cycle counter.
- `CanPendMonitorOnEvent()` returns TRUE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$I_{HSYK}$ The CTI uses the PMU as an event source.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

See also:

`PMU_HandleOverflow()`.

Chapter B12 *Debug* on page 273

## B14.7 List of supported architectural and microarchitectural events

$I_{CFXV}$ Arm recommends the use of implementation specific events based on performance behaviors of the underlying microarchitecture.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{GNJH}$ Events `0x0000-0xBFFF` are either defined or reserved for future common events. Implementations can implement any of these events.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$R_{TPLX}$ Implementations can define additional events that are specific to the implementation outside the specified reserved space.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$I_{FHHC}$ The list of common events for the PMU is as follows. The Event types are:

**Arch**

Architectural event. These events are the same across all implementations.

**uArch**

Microarchitectural event. These events might vary across different implementations.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| `0x0000` | Arch | SW_INCR | Instruction architecturally executed, condition code check pass, software increment |
| `0x0001` | uArch | L1I_CACHE_REFILL | Attributable Level 1 instruction cache refill |
| `0x0003` | uArch | L1D_CACHE_REFILL | Attributable Level 1 data cache refill |
| `0x0004` | uArch | L1D_CACHE | Attributable Level 1 data cache access |
| `0x0006` | Arch | LD_RETIRED | Instruction architecturally executed, condition code check pass, load |
| `0x0007` | Arch | ST_RETIRED | Instruction architecturally executed, condition code check pass, store |
| `0x0008` | Arch | INST_RETIRED | Instruction architecturally executed |
| `0x0009` | Arch | EXC_TAKEN | Exception taken |
| `0x000A` | Arch | EXC_RETURN | Instruction architecturally executed, condition code check pass, exception return |
| `0x000C` | Arch | PC_WRITE_RETIRED | Instruction architecturally executed, condition code check pass, software change of the PC |
| `0x000D` | Arch | BR_IMMED_RETIRED | Instruction architecturally executed, immediate branch |
| `0x000E` | Arch | BR_RETURN_RETIRED | Instruction architecturally executed, condition code check pass, procedure return |
| `0x000F` | Arch | UNALIGNED_LDST_RETIRED | Instruction architecturally executed, condition code check pass, unaligned load or store |

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| 0x0010 | uArch | BR_MIS_PRED | Mispredicted or not predicted branch speculatively executed |
| 0x0011 | uArch | CPU_CYCLES | Cycle |
| 0x0012 | uArch | BR_PRED | Predictable branch speculatively executed |
| 0x0013 | uArch | MEM_ACCESS | Data memory access |
| 0x0014 | uArch | L1I_CACHE | Attributable Level 1 instruction cache access |
| 0x0015 | uArch | L1D_CACHE_WB | Attributable Level 1 data cache write-back |
| 0x0016 | uArch | L2D_CACHE | Attributable Level 2 data cache access |
| 0x0017 | uArch | L2D_CACHE_REFILL | Attributable Level 2 data cache refill |
| 0x0018 | uArch | L2D_CACHE_WB | Attributable Level 2 data cache write-back |
| 0x0019 | uArch | BUS_ACCESS | Attributable Bus access |
| 0x001A | uArch | MEMORY_ERROR | Local memory error |
| 0x001B | uArch | INST_SPEC | Operation speculatively executed |
| 0x001D | uArch | BUS_CYCLES | Bus cycle |
| 0x001E | Arch | CHAIN | For an odd numbered counter, increment when an overflow occurs on the preceding even-numbered counter on the same PE |
| 0x001F | uArch | L1D_CACHE_ALLOCATE | Attributable Level 1 data cache allocation without refill |
| 0x0020 | uArch | L2D_CACHE_ALLOCATE | Attributable Level 2 data cache without refill |
| 0x0021 | Arch | BR_RETIRED | Instruction architecturally executed, branch |
| 0x0022 | uArch | BR_MIS_PRED_RETIRED | Instruction architecturally executed, mispredicted branch |
| 0x0023 | uArch | STALL_FRONTEND | No operation issued because of the frontend |
| 0x0024 | uArch | STALL_BACKEND | No operation issued because of the backend |
| 0x0027 | uArch | L2I_CACHE | Attributable Level 2 instruction cache access |
| 0x0028 | uArch | L2I_CACHE_REFILL | Attributable Level 2 instruction cache refill |
| 0x0029 | uArch | L3D_CACHE_ALLOCATE | Attributable Level 3 data cache allocation without refill |
| 0x002A | uArch | L3D_CACHE_REFILL | Attributable Level 3 data cache refill |
| 0x002B | uArch | L3D_CACHE | Attributable Level 3 data cache access |
| 0x002C | uArch | L3D_CACHE_WB | Attributable Level 3 data cache write-back |
| 0x0036 | uArch | LL_CACHE_RD | Last level data cache read |
| 0x0037 | uArch | LL_CACHE_MISS_RD | Last level data cache read miss |
| 0x0039 | uArch | L1D_CACHE_MISS_RD | Last level data cache read miss |
| 0x003A | uArch | OP_RETIRED | Operation retired |
| 0x003B | uArch | OP_SPEC | Operation executed |
| 0x003C | uArch | STALL | No operation sent for execution |
| 0x003D | uArch | STALL_SLOT_BACKEND | No operation sent for execution on a slot because of the backend |
| 0x003E | uArch | STALL_SLOT_FRONTEND | No operation sent for execution on a slot because of the frontend |
| 0x003F | uArch | STALL_SLOT | No operation sent for execution on a slot |

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| 0x0040 | uArch | L1D_CACHE_RD | Level 1 data cache read |
| 0x0100 | uArch | LE_RETIRED | Loop end instruction architecturally executed, entry registered in the LO_BRANCH_INFO cache |
| 0x0101 | uArch | LE_SPEC | Loop end instruction speculatively execcuted entry registered in LO_BRANCH_INFO cache |
| 0x0104 | uArch | BF_RETIRED | Branch future instruction architecturally executed, condition code check pass, and registers an entry in the LO_BRANCH_INFO cache |
| 0x0105 | uArch | BF_SPEC | Branch future instruction speculatively executed, condition code check pass, and registers an entry in the LO_BRANCH_INFOcache |
| 0x108 | uArch | LE_CANCEL | LO_BRANCH_INFO cache containing a valid loop entry cleared while not in the last iteration of the loop |
| 0x109 | uArch | BF_CANCEL | LO_BRANCH_INFO cache containing a valid BF entry cleared and associated branch not taken |
| 0x0114 | Arch | SE_CALL_S | Call to secure function, resulting in Security state change |
| 0x0115 | Arch | SE_CALL_NS | Call to non-secure function, resulting in Security state change |
| 0x0118 | Arch | DWT_CMPMATCH0 | DWT comparator 0 match |
| 0x0119 | Arch | DWT_CMPMATCH1 | DWT comparator 1 match |
| 0x011A | Arch | DWT_CMPMATCH2 | DWT comparator 2 match |
| 0x011B | Arch | DWT_CMPMATCH3 | DWT comparator 3 match |
| 0x0200 | Arch | MVE_INST_RETIRED | MVE instruction architecturally executed |
| 0x0201 | uArch | MVE_INST_SPEC | MVE instruction speculatively executed |
| 0x0204 | Arch | MVE_FP_RETIRED | MVE floating-point instruction architecturally executed |
| 0x0205 | uArch | MVE_FP_SPEC | MVE floating-point instruction speculatively executed |
| 0x0208 | Arch | MVE_FP_HP_RETIRED | MVE half-precision floating-point instruction architecturally executed |
| 0x0209 | uArch | MVE_FP_HP_SPEC | MVE half-precision floating-point instruction speculatively executed |
| 0x020C | Arch | MVE_FP_SP_RETIRED | MVE single-precision floating-point instruction architecturally executed |
| 0x020D | uArch | MVE_FP_SP_SPEC | MVE single-precision floating-point instruction speculatively executed |
| 0x0214 | Arch | MVE_FP_MAC_RETIRED | MVE floating-point multiply or multiply-accumulate instruction architecturally executed |
| 0x0215 | uArch | MVE_FP_MAC_SPEC | MVE floating-point multiply or multiply-accumulate instruction speculatively executed |
| 0x224 | Arch | MVE_INT_RETIRED | MVE integer instruction architecturally executed |
| 0x0225 | uArch | MVE_INT_SPEC | MVE integer instruction speculatively executed |
| 0x0228 | Arch | MVE_INT_MAC_RETIRED | MVE integer multiply or multiply-accumulate instruction architecturally executed |
| 0x0229 | uArch | MVE_INT_MAC_SPEC | MVE integer multiply or multiply-accumulate instruction |

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| | | | speculatively executed |
| 0x0238 | Arch | MVE_LDST_RETIRED | MVE load or store instruction architecturally executed |
| 0x0239 | uArch | MVE_LDST_SPEC | MVE load or store instruction speculatively executed |
| 0x023C | Arch | MVE_LD_RETIRED | MVE load instruction architecturally executed |
| 0x023D | uArch | MVE_LD_SPEC | MVE load instruction speculatively executed |
| 0x0240 | Arch | MVE_ST_RETIRED | MVE store instruction architecturally executed |
| 0x0241 | uArch | MVE_ST_SPEC | MVE store instruction speculatively executed |
| 0x0244 | uArch | MVE_LDST_CONTIG_RETIRED | MVE contiguous load or store instruction architecturally executed |
| 0x0245 | uArch | MVE_LDST_CONTIG_SPEC | MVE contiguous load or store instruction speculatively executed |
| 0x0248 | uArch | MVE_LD_CONTIG_RETIRED | MVE contiguous load instruction architecturally exectued |
| 0x0249 | uArch | MVE_LD_CONTIG_SPEC | MVE contiguous load instruction speculatively executed |
| 0x024C | uArch | MVE_ST_CONTIG_RETIRED | MVE contiguous store instruction architecturally executed |
| 0x024D | uArch | MVE_ST_CONTIG_SPEC | MVE contiguous store instruction speculatively executed |
| 0x0250 | uArch | MVE_LDST_NONCONTIG_RETIRED | MVE non-contiguous load or store instruction architecturally executed |
| 0x0251 | uArch | MVE_LDST_NONCONTIG_SPEC | MVE non-contiguous load or store instruction speculatively executed |
| 0x0254 | uArch | MVE_LD_NONCONTIG_RETIRED | MVE non-contiguous load instruction architecturally executed |
| 0x0255 | uArch | MVE_LD_NONCONTIG_SPEC | MVE non-contiguous load instruction speculatively executed |
| 0x0258 | uArch | MVE_ST_NONCONTIG_RETIRED | MVE non-contiguous store instruction architecturally executed |
| 0x0259 | uArch | MVE_ST_NONCONTIG_SPEC | MVE non-contiguous store instruction speculatively executed |
| 0x025C | Arch | MVE_LDST_MULTI_RETIRED | MVE memory instruction targeting multiple registers architecturally executed |
| 0x025D | uArch | MVE_LDST_MULTI_SPEC | MVE memory instruction targeting multiple registers speculatively executed |
| 0x0260 | Arch | MVE_LD_MULTI_RETIRED | MVE memory load instruction targeting multiple registers architecturally executed |
| 0x0261 | uArch | MVE_LD_MULTI_SPEC | MVE memory load instruction targeting multiple registers speculatively executed |
| 0x0264 | Arch | MVE_ST_MULTI_RETIRED | MVE memory store instruction targeting multiple registers architecturally executed |
| 0x0265 | uArch | MVE_ST_MULTI_SPEC | MVE memory store instruction targeting multiple registers speculatively executed |
| 0x028C | uArch | MVE_LDST_UNALIGNED_RETIRED | MVE unaligned memory load or store instruction architecturally executed |
| 0x028D | uArch | MVE_LDST_UNALIGNED_SPEC | MVE unaligned memory load or store |

| Event number | Event type | Event mnemonic | Description |
|---|---|---|---|
| | | | instruction speculatively executed |
| 0x0290 | uArch | MVE_LD_UNALIGNED_RETIRED | MVE unaligned load instruction architecturally executed |
| 0x0291 | uArch | MVE_LD_UNALIGNED_SPEC | MVE unaligned load instruction speculatively executed |
| 0x0294 | uArch | MVE_ST_UNALIGNED_RETIRED | MVE unaligned store instruction architecturally executed |
| 0x0295 | uArch | MVE_ST_UNALIGNED_SPEC | MVE unaligned store instruction speculatively executed |
| 0x0298 | uArch | MVE_LDST_UNALIGNED _NONCONTIG_RETIRED | MVE unaligned non-contiguous load or store instruction architecturally executed |
| 0x0299 | uArch | MVE_LDST_UNALIGNED _NONCONTIG_SPEC | MVE unaligned non-contiguous load or store instruction speculatively executed |
| 0x02A0 | Arch | MVE_VREDUCE_RETIRED | MVE vector reduction instruction architecturally executed |
| 0x02A1 | uArch | MVE_VREDUCE_SPEC | MVE vector reduction instruction speculatively executed |
| 0x02A4 | Arch | MVE_VREDUCE_FP_RETIRED | MVE floating-point vector reduction instruction architecturally executed |
| 0x02A5 | uArch | MVE_VREDUCE_FP_SPEC | MVE floating-point vector reduction instruction speculatively executed |
| 0x02A8 | Arch | MVE_VREDUCE_INT_RETIRED | MVE integer vector reduction instruction architecturally executed |
| 0x02A9 | uArch | MVE_VREDUCE_INT_SPEC | MVE integer vector reduction instruction speculatively executed |
| 0x02B8 | uArch | MVE_PRED | Cycles where one or more predicated beats architecturally executed |
| 0x02CC | uArch | MVE_STALL | Stall cycles caused by an MVE instruction |
| 0x02CD | uArch | MVE_STALL_RESOURCE | Stall cycles caused by an MVE instruction because of resource conflicts |
| 0x02CE | uArch | MVE_STALL_RESOURCE_MEM | Stall cycles caused by an MVE instruction because of memory resource conflicts |
| 0x02CF | uArch | MVE_STALL_RESOURCE_FP | Stall cycles caused by an MVE instruction because of floating-point resource conflicts |
| 0x02D0 | uArch | MVE_STALL_RESOURCE_INT | Stall cycles caused by an MVE instruction because of integer resource conflicts |
| 0x02D3 | uArch | MVE_STALL_BREAK | Stall cycles caused by an MVE chain break |
| 0x02D4 | uArch | MVE_STALL_DEPENDENCY | Stall cycles caused by MVE register dependency |
| 0x04007 | uArch | ITCM_ACCESS | Instruction TCM access |
| 0x04008 | uArch | DTCM_ACCESS | Data TCM access |
| 0x04010 | uArch | TRCEXTOUT0 | ETM external output 0 |
| 0x04011 | uArch | TRCEXTOUT1 | ETM external output 1 |
| 0x04012 | uArch | TRCEXTOUT2 | ETM external output 2 |
| 0x04013 | uArch | TRCEXTOUT3 | ETM external output 3 |
| 0x04018 | uArch | CTI_TRIGOUT4 | Cross-trigger Interface output trigger 4 |
| 0x04019 | uArch | CTI_TRIGOUT5 | Cross-trigger Interface output trigger 5 |
| 0x0401A | uArch | CTI_TRIGOUT6 | Cross-trigger Interface output trigger 6 |
| 0x0401B | uArch | CTI_TRIGOUT7 | Cross-trigger Interface output trigger 7 |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

## B14.8 Generic architectural and microarchitectural events

This section provides descriptions that apply to multiple events.

### B14.8.1 L<n>I_CACHE_REFILL (Level<n> instruction cache refill)

The counter counts each access that is counted by L<n>I_CACHE that returns instructions from beyond the Level<n> instruction cache. Beyond in this context means a Level<m> cache, where m>n, or memory.

The event indicates to software that the access missed in the Level <n> instruction cache and might have a significant performance impact because of the additional latency, compared to an access that hits in the Level <n> instruction cache.

The definition of Level<n> cache is IMPLEMENTATION DEFINED, and does not necessarily correspond to the cache levels that are defined by the CLIDR mechanism. Instead, increasing values of <n> correspond to increasing average additional latency.

The counter does not count:

- Accesses where the miss does not have a significant impact on performance.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

If the cache is shared, only accesses that are Attributable to this PE are counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.2 L<n>D_CACHE_REFILL (Level<n> data cache refill)

The counter counts each access that is counted by L<n>D_CACHE that returns data from beyond the Level<n> data cache. Beyond in this context means a Level<m> cache, where m>n, or memory.

Each access to a cache line that causes a new linefill is counted, including those from instructions that generate multiple accesses, such as load or store multiples, and `PUSH (multiple registers)` and `POP (multiple registers)` instructions. In particular, the counter counts accesses to the Level<n> cache that cause a refill. A refill includes any access that causes data to be fetched from outside the Level 1 to the Level<n> cache, even if the data is ultimately not allocated into the Level <n> cache.

The definition of Level<n> cache is IMPLEMENTATION DEFINED, and does not necessarily correspond to the cache levels that are defined by the CLIDR mechanism. Instead, increasing values of <n> correspond to increasing average additional latency.

The counter does not count:

- Accesses that do not cause a new Level<n> cache refill but are satisfied by refilling data from a previous miss.
- Accesses to a cache line that generate a memory access but not a new linefill, such as Write-Through writes that hit in the cache.
- Cache maintenance instructions.
- A write that writes an entire line to the cache and does not fetch any data from outside the Level<n> cache.
- A write that misses in the cache, and writes through the cache without allocating a line.

If the cache is shared, only accesses that are Attributable to this PE are counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.3 L<n>D_CACHE_MISS_RD (Level<n> data cache miss on read)

The counter counts each access that is counted by L<n>D_CACHE_RD that returns data from beyond the Level<n> data or unified cache. Beyond in this context means a Level<m> cache, where m>n, or memory.

The event indicates to software that the access missed in the Level<n> data or unified cache and might have a significant performance impact because of the additional latency, compared to an access that hits in the Level<n> data or unified cache.

The definition of Level<n> cache is IMPLEMENTATION DEFINED, and does not necessarily correspond to the cache levels that are defined by the CLIDR mechanism. Instead, increasing values of <n> correspond to increasing average additional latency.

The counter does not count:

- Accesses where the miss does not have a significant impact on performance.
- A miss that does not cause a new cache refill but is satisfied from a previous miss.

If the cache is shared, only accesses that are Attributable to this PE are counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.4 L<n>D_CACHE_WB (Level<n> data cache write-back)

The counter counts every write-back of data from the Level<n> data or unified cache.

The counter counts each write-back that causes data to be written from the Level<n> cache to outside of the Level<n> cache. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level<n+1> cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the Level<n> cache.
- Transfers of data from the Level<n> cache to outside of this cache that are made as a result of a coherency request. The conditions that determine which of these are counted for transfers to other Level<n> caches within the same multiprocessor cluster are IMPLEMENTATION DEFINED.

Each write-back is counted one time, even if multiple accesses are required to complete the write-back.

Whether write-backs that are made as a result of cache maintenance instructions are counted is IMPLEMENTATION DEFINED. The counter does not count:

- The invalidation of a cache line without any write-back to a Level<n+1> cache or memory.
- Writes from the PE that write through the Level<n> cache to outside of the Level<n> cache.

An Unattributable write-back event occurs when a requestor outside the PE makes a coherency request that results in write-back. If the cache is shared, then an Unattributable write-back event is not counted. If the cache is not shared, then the event is counted. It is IMPLEMENTATION DEFINED whether a write of a whole cache line that is not the result of the eviction of a line from a cache, is counted.

If the cache is shared, only accesses that are Attributable to this PE are counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.5 L<n>I_CACHE (Level<n> instruction cache access)

The counter counts each Attributable access to at least the Level<n> instruction cache. Each access to other Level<n> instruction memory structures, such as refill buffers, is also counted.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.6 L<n>D_CACHE (Level<n> data cache access)

The counter counts each Attributable memory-read or Attributable memory-write access to at least the Level<n> data or unified cache. Each access to a cache line is counted, including the multiple accesses of instructions, such as LDM or STM. Each access to other Level<n> data or unified cache memory structures is also counted.

The counter does not count cache maintenance instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### B14.8.7 L<n>D_CACHE_RD (Level<n> data cache access, read)

The counter operates the same way as L<n>D_CACHE, with the one exception that the counter counts only memory-read accesses.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

## B14.9 Common event descriptions

The common events that can be supported by the PMU counters are defined in this section. For the common features, the counters normally increment only one time for each event. The individual event descriptions include any exceptions to this. In the definitions, the term *Architecturally executed* means that the instruction flow is one where the counted instruction would have been executed in a simple sequential execution model.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### SW_INCR (0x0000, Architectural)

The counter increments on writes to the PMU_SWINC register. If the PE performs two Architecturally executed writes to the PMU_SWINC register without an intervening Context synchronization event, then the counter is incremented twice.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1I_CACHE_REFILL (0x0001, Microarchitectural)

See L<n>I_CACHE_REFILL.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE_REFILL (0x0003, Microarchitectural)

See L<n>D_CACHE_REFILL (Level<n> data cache refill).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE (0x0004, Microarchitectural)

See L<n>D_CACHE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LD_RETIRED (0x0006, Architectural)

The counter increments for every executed memory-reading instruction.

LD_RETIRED does not count the return status value of a Store-Exclusive instruction. Whether the preload instructions `PLD`, `PLDW`, and `PLI`, count as memory-reading instructions is IMPLEMENTATION DEFINED. Arm recommends that if these instruction are not implemented as NOPs, then they are counted as memory-reading instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### ST_RETIRED (0x0007, Architectural)

The counter increments for every executed memory-writing instruction. The counter does not increment for a Store-Exclusive instruction that fails.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### INST_RETIRED (0x0008, Architectural)

The counter increments for every Architecturally executed instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### EXC_TAKEN (0x0009, Architectural)

The counter increments on each exception entry. The counter does not further increment in the case of a late or derived exception, but it is incremented when tail-chaining. This corresponds to calls to the `ActivateException()` pseudocode function.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### EXC_RETURN (0x000A, Architectural)

The counter increments on each exception return. This occurs when the PE is in Handler mode, and one of the following is executed and loads an EXC_RETURN value into the PC:

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

This counter also increments on tail-chain.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### PC_WRITE_RETIRED (0x000C, Architectural)

The counter increments for every software change of the PC. This includes all of the following:

- Branch instructions.
- Loop start instructions.
- Loop end instructions.
- Memory-reading instructions that explicitly write to the PC.
- Data-processing instructions that explicitly write to the PC.

It is IMPLEMENTATION DEFINED whether the counter increments for any or all of:

- `BKPT` instructions.
- An exception generated because an instruction is UNDEFINED.
- The exception-generating instructions, `SVC`, and `UDF`.

It is IMPLEMENTATION DEFINED whether an `ISB` is counted as a software change of the PC. The counter does not increment for exceptions other than those explicitly identified in these lists.

Conditional branches are only counted if the branch is taken.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_IMMED_RETIRED (0x000D, Architectural)

The counter counts all immediate branch instructions that are Architecturally executed, which includes the immediate variants of any B branch instruction or CBNZ instruction. Conditional branches are always counted, regardless of whether the branch is taken. If an ISB is counted as a software change of the PC instruction, then it is IMPLEMENTATION DEFINED whether an ISB is counted as an immediate branch instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_RETURN_RETIRED (0x000E, Architectural)

The counter counts the following procedure return instructions:

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

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### UNALIGNED_LDST_RETIRED (0x000F, Architectural)

The counter counts each memory-reading instruction or memory-writing instruction that generates or would generate an UNALIGNED UsageFault. For instructions performing multiple memory accesses, this counter is incremented one time if at least one access generates or would generate an UNALIGNED UsageFault.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_MIS_PRED (0x0010, Microarchitectural)

The counter counts each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources, and that relates to instructions that the branch prediction resources are capable of predicting. If no program-flow prediction resources are implemented, Arm recommends that the counter counts all branches that are not taken.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### CPU_CYCLES (0x0011, Microarchitectural)

The counter increments on every cycle.

All counters are subject to changes in clock frequency, including when a WFI or WFE instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not CPU_CYCLES continues to increment when the clocks are stopped by WFI and WFE instructions.

Unlike PMU_CCNTR, this count is not affected by PMU_CTRL.DP, or by PMU_CTRL.C:

- The counter is not incremented in prohibited regions, and is not affected by PMU_CTRL.DP.
- The counter is reset when event counters are reset by PMU_CTRL.P, never by PMU_CTRL.C.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_PRED (0x0012, Microarchitectural)

The counter counts every branch or other change in the program flow that the branch prediction resources are capable of predicting. If all branches are subject to prediction, then all branches are predictable branches. If branches are decoded before the branch predictor, so that the branch prediction logic dynamically predicts only some branches, for example conditional and indirect branches, then it is IMPLEMENTATION DEFINED whether other branches are counted as predictable branches. Arm recommends that all branches are counted.

An implementation might include other structures that predict branches, such as a loop buffer that predicts short backwards direct branches as taken. Each execution of such a branch is a predictable branch. Terminating the loop might generate a misprediction event that is counted by BR_MIS_PRED. If no program-flow prediction resources are implemented, this event is optional, but Arm recommends that BR_PRED counts all branches.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MEM_ACCESS (0x0013, Microarchitectural)

The counter counts memory-read or memory-write operations that the PE made. The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these. The counter does not increment as a result of:

- Instruction memory accesses.
- Cache maintenance instructions.
- Write-back from any cache.
- Refilling of any cache.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1I_CACHE (0x0014, Microarchitectural)

See L<n>I_CACHE (Level<n> instruction cache access)

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE_WB (0x0015, Microarchitectural)

See L<n>D_CACHE_WB (Level<n> data cache write-back).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2D_CACHE (0x0016, Microarchitectural)

See L<n>D_CACHE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2D_CACHE_REFILL (0x0017, Microarchitectural)

See L<n>D_CACHE_REFILL (Level<n> data cache refill).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2D_CACHE_WB (0x0018, Microarchitectural)

See L<n>D_CACHE_WB (Level<n> data cache write-back).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BUS_ACCESS (0x0019, Microarchitectural)

The counter counts memory-read or memory-write operations that access outside of the boundary of the PE and its closely-coupled caches. This counter does not include accesses to or from a TCM. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED.

The definition of a bus access is IMPLEMENTATION DEFINED but physically it is a single request rather than a burst, (that is, for each bus cycle for which the bus is active). Bus accesses include refills of, and write-backs from, data, instruction, and unified caches. Whether bus accesses include operations that do use the bus but that do not explicitly transfer data is IMPLEMENTATION DEFINED. An Unattributable bus access occurs when a requestor outside the PE makes a request that results in a bus access, for example, a coherency request. If the bus is shared, then an Unattributable bus access is not counted. If the bus is not shared, then the event is counted.

If the bus is shared, then only Attributable bus accesses are counted. If the bus is not shared, then all bus accesses are counted.

Where an implementation has multiple buses at this boundary, this event counts the sum of accesses across all buses. If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments one time for each channel that is active on a cycle, and so it might increment by more than one in any given cycle. The maximum increment in any given cycle is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MEMORY_ERROR (0x001A, Microarchitectural)

The counter counts every occurrence of a memory error that is signaled by memory closely coupled to this PE. The definition of local memories is IMPLEMENTATION DEFINED but includes caches and tightly-coupled memories. Memory error refers to a physical error that is detected by the hardware, such as a parity or ECC error. It includes errors that are correctable and those that are not. It does not include errors that are defined in the architecture, such as MPU or SAU faults.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### INST_SPEC (0x001B, Microarchitectural)

The counter counts instructions that are speculatively executed by the PE. This includes instructions that are subsequently not architecturally executed. As a result, this event counts a larger number of instructions than the number of instructions Architecturally executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BUS_CYCLES (0x001D, Microarchitectural)

The counter increments on every cycle of the external memory interface of the PE.

If the implementation clocks the external memory interface at the same rate as the processor hardware, the counter counts every cycle.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### CHAIN (0x001E, Architectural)

Even-numbered counters never increment as a result of this event. This means the CHAIN event links the odd-numbered counter with the preceding even-numbered counter to provide a 32-bit counter.

The CHAIN event means a system can provide N 16-bit counters, N/2 32-bit counters, or a mixture of 16-bit counters and 32-bit counters. The increment of both counters is atomic with respect to software or external counter access.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE_ALLOCATE (0x001F, Microarchitectural)

The counter increments on every Attributable write that writes an entire line into the Level 1 cache without fetching from outside the Level 1 cache, for example a write from a coalescing buffer of a full cache line.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2D_CACHE_ALLOCATE (0x0020, Microarchitectural)

The counter increments on every Attributable write that writes an entire line into the Level 2 cache without fetching from outside the Level 1 or Level 2 caches, for example:

- A write-back from a Level 1 to Level 2 cache.
- A write from a coalescing buffer of a full cache line.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_RETIRED (0x0021, Architectural)

The counter counts all branches on the Architecturally executed path that would incur cost if mispredicted:

- It counts at retirement:
  - All branch instructions.
  - All memory-reading instructions that explicitly write to the PC.
  - All data-processing instructions that explicitly write to the PC.
- It counts both:
  - Branches that are taken.
  - Branches that are not taken.
- It is IMPLEMENTATION DEFINED whether this includes each of:
  - Unconditional direct branch instructions.
  - Exception-generating instructions.
  - Exception return instructions.
  - Context synchronization instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BR_MIS_PRED_RETIRED (0x0022, Microarchitectural)

The counter counts all instructions counted by BR_RETIRED that were not correctly predicted. If no program-flow prediction resources are implemented, this event counts all retired not-taken branches.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL_FRONTEND (0x0023, Microarchitectural)

The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued because there are no operations available to issue for this PE from the frontend. The division between frontend and backend is IMPLEMENTATION DEFINED. Frontend and backend events count at the same point in the pipeline.

For a simplified pipeline model of Fetch > Decode > Issue > Execute > Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.

On a given cycle, both events might be counted if the backend is unable to accept any operations and there are no operations available to issue from the frontend.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL_BACKEND (0x0024, Microarchitectural)

The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued because either:

- The backend is unable to accept any of the operations available for issue for this PE.
- The backend is unable to accept any operations.

For example, the back end might be unable to accept operations because of a resource conflict or non-availability. The division between frontend and backend is IMPLEMENTATION DEFINED. Frontend and backend events count at the same point in the pipeline.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2I_CACHE (0x0027, Microarchitectural)

See L<n>I_CACHE (Level<n> instruction cache access).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L2I_CACHE_REFILL (0x0028, Microarchitectural)

See L<n>I_CACHE_REFILL.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L3D_CACHE_ALLOCATE (0x0029, Microarchitectural)

The counter increments on every Attributable write that writes an entire line into the Level 3 cache without fetching from outside the Level 1, Level 2, or Level 3 cache, for example:

- A write-back from a Level 2 to Level 3 cache.
- A write from a coalescing buffer of a full cache line.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L3D_CACHE_REFILL (0x002A, Microarchitectural)

See L<n>D_CACHE_REFILL (Level<n> data cache refill).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L3D_CACHE (0x002B, Microarchitectural)

See L<n>D_CACHE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L3D_CACHE_WB (0x002C, Microarchitectural)

See L<n>D_CACHE_WB (Level<n> data cache write-back).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LL_CACHE_RD (0x0036, Microarchitectural)

See L<n>D_CACHE_RD (Level<n> data cache access, read).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LL_CACHE_MISS_RD (0x0037, Microarchitectural)

See L<n>D_CACHE_MISS_RD (Level<n> data cache miss on read).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE_MISS_RD (0x0039, Microarchitectural)

See L<n>D_CACHE_MISS_RD (Level<n> data cache miss on read).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### OP_RETIRED (0x003A, Microarchitectural)

The counter counts each operation counted by OP_SPEC that is later committed to the architectural state of this PE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### OP_SPEC (0x003B, Microarchitectural)

The counter counts the number of Attributable instructions or operations that are sent for execution by this PE, including those that are not committed to the architectural state of this PE.

This event might be an alias for INST_SPEC.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL (0x003C, Microarchitectural)

The counter counts every Attributable cycle on which no Attributable instruction operation was sent for execution for this PE.

The division between frontend and backend is IMPLEMENTATION DEFINED. STALL, STALL_FRONTEND and STALL_BACKEND events must count at the same point in the pipeline.

For a simplified pipeline model of Fetch, Decode, Issue, Execute, Retire, Arm recommends that the events are counted when instructions are dispatched from Decode to Issue.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL_SLOT_BACKEND (0x003D, Microarchitectural)

Counts each slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution because either:

- The backend was unable to accept the instruction operation available for this PE on the slot
- The backend is unable to accept any operations on the slot.

The division between frontend and backend is IMPLEMENTATION DEFINED.
STALL_SLOT, STALL_SLOT_FRONTEND and STALL_SLOT_BACKEND events count at the same point in the pipeline.

On a given cycle, both the STALL_SLOT_FRONTEND and the STALL_SLOT_BACKEND event might be counted if the backend is unable to accept any operations and there are no operations available to issue from the frontend.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL_SLOT_FRONTEND (0x003E, Microarchitectural)

Counts each slot counted by STALL_SLOT where no Attributable instruction or operation was sent for execution because there was no Attributable instruction or operation available to issue for this PE from the frontend for the slot.

The division between frontend and backend is IMPLEMENTATION DEFINED.
STALL_SLOT, STALL_SLOT_FRONTEND and STALL_SLOT_BACKEND events count at the same point in the pipeline.

On a given cycle, both STALL_SLOT_FRONTEND and STALL_SLOT_BACKEND event might be counted if the backend is unable to accept any instructions operations and there are no instructions or operations available to issue from the frontend.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### STALL_SLOT (0x003F, Microarchitectural)

The counter counts on each Attributable cycle the number of instruction or operation slots that were not occupied by an instruction or operation Attributable to this PE.

The definition of a slot is IMPLEMENTATION DEFINED, but there is a fixed number of slots, WIDTH, that are available on each cycle, so that the formula STALL_SLOT / (CPU_CYCLES x WIDTH) gives the utilization of the slots of the processor by Attributable instruction or operations of this PE. Each slot can hold at most one instruction or operation each cycle.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### L1D_CACHE_RD (0x0040, Microarchitectural)

See L<n>D_CACHE_RD (Level<n> data cache access, read).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LE_RETIRED (0x0100, Microarchitectural)

The counter increments for every Architecturally executed loop end `LE, LETP` instruction, when that instruction registers an entry inside the LO_BRANCH_INFO cache.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LE_SPEC (0x0101, Microarchitectural)

The counter increments for every speculatively executed loop end `LE, LETP` instruction, when that instruction registers an entry inside the LO_BRANCH_INFO cache.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BF_RETIRED (0x0104, Microarchitectural)

The counter increments for every Architecturally executed branch future instruction, when that instruction registers an entry inside the LO_BRANCH_INFO cache.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BF_SPEC (0x0105, Microarchitectural)

The counter increments for every speculatively executed branch future instruction, when that instruction registers an entry inside the LO_BRANCH_INFO cache.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### LE_CANCEL (0x0108, Microarchitectural)

The LO_BRANCH_INFO cache was cleared while it contained a valid loop entry as set up by a loop end instruction, and that did not coincide with the last iteration of the loop.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### BF_CANCEL (0x0109, Microarchitectural)

The LO_BRANCH_INFO cache was cleared while it contained a valid branch entry as set up by a branch future instruction, and that did not coincide with a taken implicit branch.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### SE_CALL_S (0x0114, Architectural)

The counter increments for every Architecturally executed SG instruction that results in a Security state transition from Non-secure state to Secure state.

Arm recommends that this counter increments regardless of the Security state filters that are applied to the PMU.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### SE_CALL_NS (0x0115, Architectural)

The counter increments for every Architecturally executed BLXNS instruction that results in a Security state transition from Secure state to Non-secure state.

Arm recommends that this counter increments regardless of the Security state filters that are applied to the PMU.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### DWT_CMPMATCH<n> (0x0118, Architectural)

**Where <n> = 0-3.**

The counter increments for each successful comparator match indicated by DWT comparator <n>. The comparator does not increment if the comparator is not implemented or is disabled.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **DSPDE**.*

### MVE_INST_RETIRED (0x0200, Architectural)

The counter increments for each Architecturally executed MVE instruction that is subject to beat-wise execution. This includes instructions that are partially or fully predicated, and instructions that resume execution after returning from an exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_INST_SPEC (0x0201, Microarchitectural)

The counter increments for each speculatively executed MVE instruction that is subject to beat-wise execution. This includes instructions that are subsequently not Architecturally executed. As a result, this event might count a larger number of instructions than the number of instructions Architecturally executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_RETIRED (0x0204, Architectural)

The counter increments for each Architecturally executed MVE instruction, as counted by MVE_INST_RETIRED, that operates on floating-point data. This includes MVE conversion instructions that convert to or from floating-point representation, as well as floating-point vector compare and vector predicate instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_SPEC (0x0205, Microarchitectural)

The counter increments for each speculatively executed MVE instruction, as counted by MVE_INST_SPEC, that operates on floating-point data. This includes MVE conversion instructions that convert to or from floating-point representation, as well as floating-point vector compare and vector predicate instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_HP_RETIRED (0x0208, Architectural)

The counter increments for each Architecturally executed MVE floating-point instruction that operates on floating-point data, as counted by MVE_FP_RETIRED, and which additionally operates on half-precision data. This includes MVE conversion instructions that convert to or from half-precision floating-point representation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_HP_SPEC (0x0209, Microarchitectural)

The counter increments for each speculatively executed MVE floating-point instruction that operates on floating-point data, as counted by MVE_FP_SPEC, and which additionally operates on half-precision data. This includes MVE conversion instructions that convert to or from half-precision floating-point representation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_SP_RETIRED (0x020C, Architectural)

The counter increments for each Architecturally executed MVE floating-point instruction that operates on floating-point data, as counted by MVE_FP_RETIRED, and which additionally operates on single-precision data. This does not include MVE conversion instructions that convert to or from single-precision floating-point representation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_SP_SPEC (0x020D, Microarchitectural)

The counter increments for each speculatively executed MVE floating-point instruction that operates on floating-point data, as counted by MVE_FP_SPEC, and which additionally operates on single-precision data. This does not include MVE conversion instructions that convert to or from single-precision floating-point representation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_MAC_RETIRED (0x0214, Architectural)

The counter increments for each Architecturally executed MVE floating-point instruction that performs a multiply or multiply-accumulate operation. This includes instructions that perform fused multiply-accumulation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_FP_MAC_SPEC (0x0215, Microarchitectural)

The counter increments for each speculatively executed MVE floating-point instruction that performs a multiply or multiply-accumulate operation. This includes instructions that perform fused multiply-accumulation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_INT_RETIRED (0x0224, Architectural)

The counter increments for each Architecturally executed MVE beat-wise integer or fixed-point instruction. This does not include any floating-point conversion instructions. This does include integer vector compare and vector predicate instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_INT_SPEC (0x0225, Microarchitectural)

The counter increments for each speculatively executed MVE beat-wise integer or fixed-point instruction. This does not include any floating-point conversion instructions. This does include integer vector compare and vector predicate instructions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_INT_MAC_RETIRED (0x0228, Architectural)

The counter increments for each Architecturally executed MVE integer or fixed-point instruction, as counted by MVE_INT_RETIRED, which additionally performs a multiply or multiply-accumulate operation. This includes reducing variants.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_INT_MAC_SPEC (0x0229, Microarchitectural)

The counter increments for each speculatively executed MVE integer or fixed-point instruction, as counted by MVE_INST_SPEC, which additionally performs a multiply or multiply-accumulate operation. This includes reducing variants.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_RETIRED (0x0238, Architectural)

The counter increments for each Architecturally executed MVE load instruction that writes data to a Q register or store instruction that sources data from a Q register.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_SPEC (0x0239, Microarchitectural)

The counter increments for each speculatively executed MVE load instruction that writes data to a Q register or store instruction that sources data from a Q register.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_RETIRED (0x023C, Architectural)

The counter increments each time MVE_LDST_RETIRED increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_SPEC (0x023D, Microarchitectural)

The counter increments each time MVE_LDST_SPEC increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_RETIRED (0x0240, Architectural)

The counter increments each time MVE_LDST_RETIRED increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_SPEC (0x0241, Microarchitectural)

The counter increments each time MVE_LDST_SPEC increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_CONTIG_RETIRED (0x0244, Microarchitectural)

The counter increments each time a contiguous memory load or store instruction is Architecturally executed, and results in an optimal number of memory accesses. The counter increments for scatter-gather instructions that are promoted to contiguous accesses. The counter increments regardless of alignment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_CONTIG_SPEC (0x0245, Microarchitectural)

The counter increments each time a contiguous memory load or store instruction is speculatively executed, and results in an optimal number of memory accesses. The counter increments for scatter-gather instructions that are promoted to contiguous accesses. The counter increments regardless of alignment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_CONTIG_RETIRED (0x0248, Microarchitectural)

The counter increments each time MVE_LDST_CONTIG_RETIRED increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_CONTIG_SPEC (0x0249, Microarchitectural)

The counter increments each time MVE_LDST_CONTIG_SPEC increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_CONTIG_RETIRED (0x024C, Microarchitectural)

The counter increments each time MVE_LDST_CONTIG_RETIRED increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_CONTIG_SPEC (0x024D, Microarchitectural)

The counter increments each time MVE_LDST_CONTIG_SPEC increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_NONCONTIG_RETIRED (0x0250, Microarchitectural)

The counter increments each time a non-contiguous memory load or store instruction is Architecturally executed, and results in an increased number of accesses compared to a contiguous operation. The counter is incremented one time, regardless of the additional number of memory requests that are required to complete the execution of that instruction. The counter does not increment if a scatter-gather instruction is promoted to contiguous memory accesses.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_NONCONTIG_SPEC (0x0251, Microarchitectural)

The counter increments each time a non-contiguous memory load or store instruction is speculatively executed, and results in an increased number of accesses compared to a contiguous operation. The counter is incremented one time, regardless of the additional number of memory requests that are required to complete the execution of that instruction. The counter does not increment if a scatter-gather instruction is promoted to contiguous memory accesses.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_NONCONTIG_RETIRED (0x0254, Microarchitectural)

The counter increments each time MVE_LDST_NONCONTIG_RETIRED increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_NONCONTIG_SPEC (0x0255, Microarchitectural)

The counter increments each time MVE_LDST_NONCONTIG_SPEC increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_NONCONTIG_RETIRED (0x0258, Microarchitectural)

The counter increments each time MVE_LDST_NONCONTIG_RETIRED increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_NONCONTIG_SPEC (0x0259, Microarchitectural)

The counter increments each time MVE_LDST_NONCONTIG_SPEC increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_MULTI_RETIRED (0x025C, Architectural)

The counter increments whenever a VLD2x, VST2x, VLD4x, or VST4x instruction is Architecturally executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_MULTI_SPEC (0x025D, Microarchitectural)

The counter increments whenever a VLD2x, VST2x, VLD4x, or VST4x instruction is speculatively executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_MULTI_RETIRED (0x0260, Architectural)

The counter increments whenever a VLD2x, or VLD4x instruction is Architecturally executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_MULTI_SPEC (0x0261, Microarchitectural)

The counter increments whenever a VLD2x, or VLD4x instruction is speculatively executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_MULTI_RETIRED (0x0264, Architectural)

The counter increments whenever a VST2x, or VST4x instruction is Architecturally executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_MULTI_SPEC (0x0265, Microarchitectural)

The counter increments whenever a VST2x, or VST4x instruction is speculatively executed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_UNALIGNED_RETIRED (0x028C, Microarchitectural)

The counter increments when MVE_LDST_RETIRED increments as a result of an instruction that requires one or more additional memory accesses because of misalignment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_UNALIGNED_SPEC (0x028D, Microarchitectural)

The counter increments when MVE_LDST_SPEC increments as a result of an instruction that requires one or more additional memory accesses because of the misalignment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_UNALIGNED_RETIRED (0x0290, Microarchitectural)

The counter increments when MVE_LDST_UNALIGNED_RETIRED increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LD_UNALIGNED_SPEC (0x0291, Microarchitectural)

The counter increments when MVE_LDST_UNALIGNED_SPEC increments as a result of a load operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_UNALIGNED_RETIRED (0x0294, Microarchitectural)

The counter increments when MVE_LDST_UNALIGNED_RETIRED increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_ST_UNALIGNED_SPEC (0x0295, Microarchitectural)

The counter increments when MVE_LDST_UNALIGNED_SPEC increments as a result of a store operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_UNALIGNED_NONCONTIG_RETIRED (0x0298, Microarchitectural)

This counter increments whenever both MVE_LDST_UNALIGNED_RETIRED and MVE_LDST_NONCONTIG_RETIRED increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_LDST_UNALIGNED_NONCONTIG_SPEC (0x0299, Microarchitectural)

This counter increments whenever both MVE_LDST_UNALIGNED_SPEC and MVE_LDST_NONCONTIG_SPEC increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_RETIRED (0x02A0, Architectural)

The counter increments whenever an MVE instruction that operates on a vector to produce a scalar result that is stored in a general-purpose result is Architecturally executed. This includes only instructions that have the ‘V’ suffix.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_SPEC (0x02A1, Microarchitectural)

The counter increments whenever an MVE instruction that operates on a vector to produce a scalar result that is stored in a general-purpose register is speculatively executed. This includes only instructions that have the 'V' suffix.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_FP_RETIRED (0x02A4, Architectural)

The counter increments whenever both MVE_VREDUCE_RETIRED and MVE_FP_RETIRED increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_FP_SPEC (0x02A5, Microarchitectural)

The counter increments whenever both MVE_VREDUCE_SPEC and MVE_FP_SPEC increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_INT_RETIRED (0x02A8, Architectural)

The counter increments whenever both MVE_VREDUCE_SPEC and MVE_INST_RETIRED increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_VREDUCE_INT_SPEC (0x02A9, Microarchitectural)

The counter increments whenever both MVE_VREDUCE_SPEC and MVE_INST_SPEC increment.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_PRED (0x02B8, Microarchitectural)

The counter increments on each Architecture tick where one or more beats of an MVE beat-wise instruction architecturally completes, and where one or more of these beats is partially or fully predicated false by VPR or loop-tail predication.

If a beat is interrupted by an exception and does not architecturally complete, or if the beat is masked by EPSR.ECI when resuming execution following an exception, Arm recommends that the counter does not increment.

The ratio (BEATS_PER_TICK) / (4 * then offers an approximate insight into the proportion of MVE instructions affected by predication, where BEATS_PER_TICK is an IMPLEMENTATION DEFINED average number of beats, including from distinct overlapping instructions, executed per Architecture tick.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL (0x02CC, Microarchitectural)

The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued as a direct result of an MVE instruction that is either currently executing or attempting to execute.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_RESOURCE (0x02CD, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to an MVE instruction failing to execute because there are no available resources in the PE capable of executing that instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_RESOURCE_MEM (0x02CE, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to an MVE instruction failing to execute because there are no available memory resources in the PE capable of executing that instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_RESOURCE_FP (0x02CF, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to an MVE instruction failing to execute because there are no available floating-point resources in the PE capable of executing that instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_RESOURCE_INT (0x02D0, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to an MVE instruction failing to execute because there are no available integer resources in the PE capable of executing that instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_BREAK (0x02D3, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to waiting for the completion of an in-flight MVE instruction. A possible example is when an MVE chainable instruction completes before executing a scalar instruction. Arm recommends that the counter increments only if no other specific attributable cause can be identified.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### MVE_STALL_DEPENDENCY (0x02D4, Microarchitectural)

The counter increments whenever MVE_STALL increments, and where the cause is attributable to the subsequent instruction being delayed to resolve a register RAW conflict.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### ITCM_ACCESS (0x4007, Microarchitectural)

The counter counts memory read or memory write operations that the PE made to an instruction TCM.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### DTCM_ACCESS (0x4008, Microarchitectural)

The counter counts memory read or memory write operations that the PE made to a data or unified TCM.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

### TRCEXTOUT<n> (0x4010, Microarchitectural)

**Where <n> = 0-3.**

The counter counts for each event signaled by the ETM external event <n>.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **DSPDE**.*

### CTI_TRIGOUT<n> (0x4018, Microarchitectural)

**Where <n> = 4-7.**

The counter counts for each event signaled by the CTI output trigger <n>.

Note: CTI output triggers are input events to the PMU and PE Trace Unit.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU** && **DSPDE**.*

## B14.10 Required PMU events

$R_{ZLFR}$ The architecture requires that the PMU supports at least the following common events:

- `0x000` SW_INCR:
  - Instruction Architecturally executed, condition code check pass, software increment.
- `0x003` L1D_CACHE_REFILL:
  - Attributable Level 1 data cache refill.
  - This event is only required if the implementation includes a Level 1 data or unified cache.
- `0x004` L1D_CACHE:
  - Attributable Level 1 data cache access.
  - This event is only required if the implementation includes a Level 1 data or unified cache.
- `0x022` BR_MIS_PRED_RETIRED:
  - Instruction Architecturally executed for a mispredicted branch.
  - This event is only required if the implementation includes program flow prediction.
- `0x011` CPU_CYCLE:
  - Cycle.
- `0x012` BR_PRED:
  - Predictable branch speculatively executed.
  - This event is only required if the implementation includes program flow prediction.
- `0x008` INST_RETIRED:
  - Instruction Architecturally executed.
- `0x023` STALL_FRONTEND:
  - No operation issued because of the frontend.
- `0x024` STALL_BACKEND:
  - No operation issued because of the backend.
- `0x200` MVE_INST_RETIRED:
  - MVE instruction Architecturally executed.
  - This event is only required if the implementation includes MVE.
- `0x238` MVE_LDST_RETIRED:
  - MVE load or store instruction Architecturally executed.
  - This event is only required if the implementation includes MVE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

$I_{CVSG}$ Arm recommends that events `0x022` BR_MIS_PRED_RETIRED and `0x012` BR_PRED are implemented as described in $R_{ZLFR}$.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **PMU**.*

## B14.11 IMPLEMENTATION DEFINED event numbers

$R_{ZJSZ}$ For IMPLEMENTATION DEFINED event numbers, each counter is independently defined to either:

- Increment only one time for each event.
- Count the duration for which an event occurs.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***PMU****.*

$I_{LCNR}$ Arm recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions and common count numbers applied to all of their implementations. In general, Arm recommends standardization across implementations with common features. However, Arm recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***PMU****.*

$I_{RKHN}$ Arm strongly recommends that the IMPLEMENTATION DEFINED events allow the user to measure the utilization of any microarchitectural features that the implementation considers important for system performance, and any significant deviations from optimal performance.

*Applies to an implementation of the architecture from* ***Armv8.1-M*** *onwards. The extension requirements are -* ***PMU****.*

# Chapter B15
# Reliability, Availability, and Serviceability (RAS) Extension

This chapter specifies the Armv8.1-M Reliability, Availability, and Serviceability (RAS) Extension. A minimal implementation of this is mandatory for any implementation of the Armv8.1-M architecture, but any additional RAS features are optional.

This chapter contains the following sections:



*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.1 Overview

$I_{ZTMS}$ For a detailed description of possible RAS errors, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile*.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TXGN}$ A minimum implementation of the RAS Extension is required for an implementation of the v8.1-M architecture.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{RSMP}$ ID_PFR0.RAS is nonzero when the RAS Extension is implemented.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{CTQJ}$ An error is a deviation from correct service. For the purpose of describing the RAS Extension, deviation from correct service is defined using the following terms:

- A *failure* is the event of deviation from correct service. This includes data corruption, data loss, and service loss.
- An *error* is the deviation.
- A *fault* is the cause of the error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{FCTN}$ When a PE accesses memory or other state, an error might be detected in that memory or state, and corrected, deferred, or signaled to the PE as a Detected error. It is IMPLEMENTATION DEFINED whether an error that is detected by a consumer of a write from a PE is signaled to the PE and becomes a Detected error that is consumed by the PE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{LTTV}$ The nodes that are included as part of a PE, including an Armv8-M PE, are IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TLFS}$ A single error might generate multiple exceptions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{NJVH}$ Software must be aware that errors might be double-reported.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

See also:

B15.9 *Minimal RAS implementation* on page 418.

## B15.2 Taxonomy of errors

$I_{GNNQ}$ The architecture does not specify techniques for:

- Fault prevention.
- Fault removal.
- Fault injection.
- Testing.

These are IMPLEMENTATION DEFINED and are outside the scope of this architecture.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

See also:

- *Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.*

### B15.2.1 Architectural error propagation

$R_{MVFK}$ For a PE, Error propagation applies to the propagation of detected errors between the general-purpose registers or the Floating-point Extension register file, and any program-visible architectural state of the PE, including:

- Other general-purpose registers and the Floating-point Extension register file.
- Memory-mapped registers.
- Special-purpose registers.
- Memory.

That is, the error is propagated by:

- A store of a corrupt value.
- A write of a corrupt value to a System register or a special-purpose register. Infecting a System register state might mean that the PE generates transactions that would not otherwise be permitted.
- Any operation that would not have been permitted to occur had the error not been activated, including:
  - A load or instruction fetch that would not have been permitted, including those from hardware speculation or prefetching.
  - A store to an incorrect address or a store that would not have been made or not permitted.
  - A direct or indirect write to a special-purpose or System register that would not have been made or not permitted.
  - Assertion of any signal, such as an interrupt, that would not have been asserted.
- Any operation not occurring that would have occurred had the error not been activated.
- Taking an asynchronous exception.
- The PE discarding data that it holds in a modified state.
- Any other loss of uniprocessor semantics, ordering or coherency.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{RGPS}$ The propagated error is silently propagated if it is not signaled to a consumer as a Detected error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JKVQ}$ The features that a PE includes to contain an error are IMPLEMENTATION DEFINED, and it is IMPLEMENTATION DEFINED whether an error can be signaled to the consumer as a Detected error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

See also:

*Arm® Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for the Armv8-A architecture profile.*

### B15.2.2 Architecturally infected, contained, and uncontained

$R_{VKZT}$ Infected, Poisoned, Containable and Uncontainable apply to all program-visible architectural state of the PE, including general-purpose registers, the Floating-point Extension register file, special purpose registers, System registers, and memory.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{ZPCB}$ An error is Uncontainable by the PE if the error is silently propagated, unless it is contained because all of the following are true:

- The corrupt value is in the general-purpose register or in the Floating-point Extension register file.
- The error has only been silently propagated by an instruction that occurs in program order after one of the following:
  - Taking a BusFault that is generated by the error.
  - An Error Synchronization Barrier operation that synchronizes the error.
- The error is not silently propagated in any other way.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

### B15.2.3 Architecturally consumed errors

$R_{RNNF}$ For a PE, an error is architecturally consumed if any of the following are true:

- An instruction commits the corruption into the visible state of the PE.
- The error is on an instruction fetch and the instruction is committed for execution.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{MKBF}$ The PE takes action for a detected, architecturally consumed error either by:

- Generating an error exception.
- Entering a failure mode.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

### B15.2.4 Other errors

$I_{TCZS}$ Errors from software faults are outside the scope of the RAS Extension error recovery architecture.

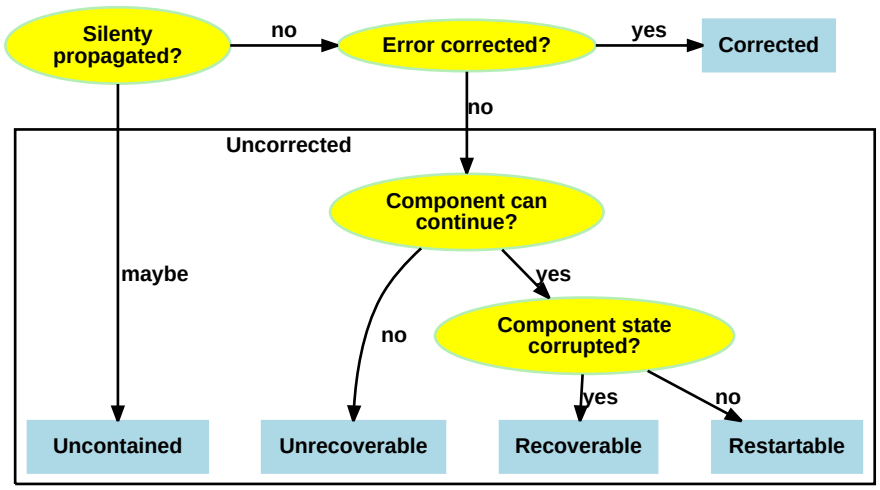
*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{KDJP}$ From within the PE itself, other errors might be detected. These are not errors that are detected by the architectural model of the PE and so are treated like errors that are detected by another component. Other components might report errors to a PE using error recovery interrupts. An example of this is when the cache, not the PE, detects a RAM error. Other components might report errors to a PE using error recovery interrupts.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.3 Generating error exceptions

$I_{WZHQ}$ The following diagram shows the taxonomy of consumed errors.




*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{PVJQ}$ An error exception is generated for all detected RAS errors that are neither Corrected nor Deferred errors. These error exceptions are signaled to, and consumed by, a PE and are not silently propagated.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{RPFG}$ A Corrected error is detected and corrected by the PE, and is not silently propagated.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{NDMC}$ In normal circumstances a Corrected error no longer infects the node. In IMPLEMENTATION DEFINED circumstances, the Corrected error might remain latent in the node.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{PSKL}$ For an Uncontainable error, if the error cannot be isolated to an application, the system must be shut down by software to avoid Catastrophic failure.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{LFCG}$ For an Unrecoverable error, the application cannot continue and must be isolated by software methods.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{PTXC}$ For a Recoverable error, if software cannot locate and repair the error, the application must be isolated by software methods.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{RSJP}$ For a Restartable error, software might take action to locate and repair the error before it is consumed. The PE can be restarted by software without software taking any action to locate and repair the error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{SJKL}$ On each error exception, it is IMPLEMENTATION DEFINED whether the error has been contained or whether it is Uncontainable. If the error has been contained, it is further IMPLEMENTATION DEFINED whether the state of the PE on taking the error exception is Unrecoverable, Recoverable, or Restartable:

**Uncontainable error** (UC): The error is Uncontainable if it has been, or might have been, silently propagated. This is also referred to as an Uncontained error.

**Unrecoverable error** (UEU): The state of the PE is Unrecoverable if all of the following are true:

- The error has not been silently propagated.
- The PE cannot recover execution from the return address of the exception. This might be because of one of the following:
  - The error has been architecturally consumed by the PE and infected the state of the PE general-purpose registers, the Floating-point Extension register file, and System registers.
  - The exception is asynchronous.

**Recoverable error** (UER): The state of the PE is Recoverable if all of the following are true:

- The error has not been silently propagated.
- The error has not been architecturally consumed by the PE (the architectural state of the PE is not infected).
- The exception is synchronous and the PE can recover execution from the return address of the exception.

The PE cannot make correct progress without either consuming the error or otherwise making the error Unrecoverable. The error remains latent in the system.

**Restartable error** (UEO): The state of the PE is Restartable if all of the following are true:

- The error has not been silently propagated.
- The error has not been architecturally consumed by the PE (the architectural state of the PE is not infected).
- The exception is synchronous and the PE can recover execution from the return address of the exception.

The PE can make progress. However, the error might remain latent in the system.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{KZPT}$ The set of error types that can be reported by an implementation is IMPLEMENTATION DEFINED. An implementation can report:

- Any Restartable error as any of Recoverable, Unrecoverable, or Uncontainable.
- Any Recoverable error as either Unrecoverable or Uncontainable.
- Any Unrecoverable error as Uncontainable.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{XCSK}$ If the state of the PE is reported as Recoverable, this does not mean that the error can be recovered from. For example, because the error in memory might be one which does not allow software to recover the operation. Rather, software *might* be able to recover if it can repair the error and continue.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{SGHS}$ A bus error might be raised in response to:

- An architectural memory read, or reads from instruction fetches.
- An architectural write to memory, or cache maintenance operation.
- A read from memory because of hardware speculation, prefetching, or other non-architectural mechanisms.

The events that trigger bus errors are IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{QLCQ}$ It is IMPLEMENTATION DEFINED whether a RAS error that is detected by the consumer of a write from a PE:

- Is deferred to the consumer.

- Is returned to the PE as a bus error.
- Generates an error recovery interrupt.

The behavior might vary by physical address or memory type.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TFLL}$ The method by which the error is deferred depends on the component and is implementation-specific.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{RDQV}$ A common mechanism to defer the error is to create Poisoned state, which subsequently generates an error when that state is accessed.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{CXSW}$ The size of the Protection granule for any implemented error detection mechanism is IMPLEMENTATION DEFINED, and a system might implement multiple error detection mechanisms with different Protection granule sizes.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{GBDR}$ The mechanisms for clearing an error or poison from a Protection granule is IMPLEMENTATION DEFINED, and it is IMPLEMENTATION DEFINED whether any such mechanism exists.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TTQG}$ A BusFault exception is a synchronous recoverable or restartable error exception that is generated by instruction fetches or architectural memory accesses.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{PXMR}$ BFSR is populated for every BusFault that is generated by a RAS error. RFSR is populated for every RAS BusFault exception, and RFSR.V is set.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{FZKC}$ The severity of the error and the state of the PE are reported when the error exception is pended.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{ZKNJ}$ If an error exception occurs while the BusFault exception handler is handling a previous error exception, the existing exception nesting behaviors apply. This might cause the exception to escalate to a HardFault or Lockup.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{RSSQ}$ A synchronous RAS error that was caused by a DAP request results in an error being returned to the debugger. An asynchronous RAS error will succeed, but for a read access *unknown* data is returned. No BusFault exception is raised and RFSR, BFAR, and BFSR are unchanged.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

### B15.3.1 Error correction and deferment

$R_{WXZD}$ Hardware corrects or defers an error if it can do so. The error is logged, and a fault handling interrupt is generated for fault handling purposes if the node is configured to do so.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.4 Error Synchronization Barrier (ESB)

$I_{JLPC}$ The RAS Extension introduces a new instruction, the Error Synchronization Barrier (ESB).

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{DNTZ}$ ESB acts as a NOP when the system cannot synchronize RAS errors.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{DLKL}$ ESB does not itself update any registers, but any ensuing BusFault exception will update at least BFSR and RFSR.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{LNRD}$ ESB might update syndrome bits, for example if IESBs are disabled.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{ZLCQ}$ If there is a pending BusFault, which might have been forced to be recognized as a direct result of the call to `SynchronizeBusFault()`, then an ESB also acts as a Data Synchronization Barrier so that a subsequent load of the memory mapped syndrome registers and bits in BFSR, RFSR and SHCSR is guaranteed to return the correct values.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

See also:

B15.5 *Implicit Error Synchronization (IESB)* on page 410.

### B15.4.1 ESB and Unrecoverable errors

$R_{VDCF}$ An ESB acts as a barrier to all unrecoverable (RAS-related and non-RAS related) bus errors and causes any Latent faults to be recognized synchronously with the instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{MSMM}$ Depending on the current state of the PE, recognizing any latent bus errors might simply result in a BusFault being pended, or the BusFault might be taken or escalated.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{GCBZ}$ The Error Synchronization Barrier operation contains the error for the current software context.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

### B15.4.2 ESB and other containable errors

$I_{PHSB}$ For other types of Containable error:

- A Recoverable error has not yet been consumed by the PE.
- Restartable and Corrected errors, and BusFault exceptions from reads by hardware speculation that do not corrupt the state of the PE, have not been consumed by the PE.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{WKMP}$ An unconsumed RAS error might be taken at the ESB.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

### B15.4.3 ESB and other errors

$R_{PHJW}$ Synchronous BusFault exceptions are not synchronized by an Error Synchronization Barrier.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TXBQ}$ Interrupts are not synchronized by an ESB.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{VSDB}$ An ESB instruction will synchronize asynchronous BusFaults.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{WRJG}$ An ESB always synchronizes Containable errors, but it is IMPLEMENTATION DEFINED whether IMPLEMENTATION DEFINED and uncategorized BusFault exceptions are Uncontainable, and whether they can be synchronized by an Error Synchronization Barrier.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{TZTR}$ Uncontainable errors might not have been contained, and Uncontainable BusFault exceptions might be asynchronous. An Uncontainable error might be taken at the ESB but this is not architecturally required.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TSGR}$ It is IMPLEMENTATION DEFINED whether IMPLEMENTATION DEFINED and uncategorized interrupts are Containable or Uncontainable.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{HGSQ}$ An Uncontainable error might be taken at an Error Synchronization Barrier or recorded in RFSR.IS by an instr.ESB]ESB instruction.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.5 Implicit Error Synchronization (IESB)

$I_{SSXC}$ To ensure that faults arise in the appropriate PE state, an implicit error synchronization event (IESB) can optionally be inserted on every exception entry, exception return, and on lazy state stacking operations.

Neither entry to, nor exit from, Debug state insert an IESB.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{ZGNM}$ IESB only updates at least one of BFSR or RFSR if there was a latent asynchronous bus error that was synchronized.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{VTKM}$ An IESB acts as a barrier to all Unrecoverable (RAS-related and non-RAS related) bus errors.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{GRKC}$ Enabling IESBs causes all asynchronous BusFaults to escalate as if they were synchronous BusFaults, regardless of whether they were asynchronously recognized, or forced to be recognized by an ESB or IESB.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{NGQC}$ Asynchronous BusFaults that escalate synchronously because IESBs are enabled are still reported as asynchronous faults. This means that the exception return address does not point to the instruction that caused the fault.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{WRDB}$ AIRCR.IESB determines whether an exception entry or an exception return behaves as an implicit error synchronization event and requires any outstanding RAS exceptions to be acknowledged:

- On exception entry, only errors relating to the background code and register stacking are acknowledged.
- On exception return, only errors relating to the handler code are acknowledged. Register unstacking relates to the background code and is not acknowledged.

This ensures that the fault is associated with the context that triggered it. Any resulting BusFault is handled using the existing derived exception rules.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{FTLD}$ Bus errors that are raised by an IESB on exception entry or exception return are handled in the same way as derived faults that occur as a result of exception stacking and unstacking.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{PDXH}$ Arm recommends that implementations support the implicit error synchronization behavior when asynchronous exceptions and associated timings create the possibility of errors being associated with a context that is different to the one that caused the error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{WJNK}$ Writes by software to AIRCR.IESB can be ignored by the PE if it does not support configurable implicit ESB insertion. The value that is read back determines whether the feature is supported.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{LZWF}$ When IESBs are enabled:

- An IESB occurs prior to stacking a lazy floating-point context, and any RAS errors are associated with foreground code.
- An IESB occurs after stacking a lazy floating-point context, and any RAS errors are associated with the register stacking.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JMXC}$ RAS faults set the first applicable syndrome information from the following list:

- BFSR.LSPERR if the error is attributable to lazy state preservation stacking.
- BFSR.STKERR if the error is attributable to exception register stacking.
- BFSR.UNSTKERR if the error is attributable to exception register unstacking.
- BFSR.IBUSERR if the error is caused by an instruction fetch or prefetch.
- BFSR.PRECISERR if the error occurred synchronously and is attributable to instruction execution.
- BFSR.IMPRECISERR otherwise.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{DHDZ}$ Arm recommends that an Implicit Error Synchronization Barrier is used on changes of Security state.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.6 Fault handling

$R_{QRVP}$ There are four forms of RAS error reporting:

- A bus error in response to an action from the PE.
- An error recovery interrupt.
- A Fault handling interrupt.
- A critical error interrupt.

Depending on the system configuration, the interrupts can be routed through a system interrupt controller or the NVIC.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{WPNL}$ An error recovery interrupt can notify software of RAS faults that are detected in the system.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{LVWP}$ A fault handling interrupt can notify software of RAS events that are detected in the system.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JLPR}$ When an error is detected by a node that supports fault reporting, the node records the error in Error record registers and generates a RAS fault handling interrupt, if configured to do so.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{XKVL}$ The RAS fault handling interrupt might be routed at the system level to a PE that is not directly affected by the fault. Conversely, the PE might receive fault handling interrupts relating to other devices in the system.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JTXP}$ RAS fault handling interrupts might be sent to a dedicated fault handling PE by IMPLEMENTATION DEFINED means.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{SMXP}$ The error recovery interrupts, fault handling interrupts, and critical error interrupts, are level sensitive or pulse sensitive in the same way as other interrupts. It is IMPLEMENTATION DEFINED whether a node employs level sensitive or pulse sensitive interrupts.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{GTMF}$ The error recovery interrupts, fault handling interrupts, and critical error interrupts are pended in finite times after changes to the error conditions or to the fault being observed or corrected.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{NHGH}$ Support for critical error conditions and critical error interrupts at a node is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{KLPK}$ An example of a critical error is one where the node has entered a failure mode, which means that the primary error recovery mechanisms cannot be used. For example, if a memory controller enters a failure mode and stops handling memory requests from application processors, and application processors host the primary error recovery software, then the error is signaled to a secondary error controller that has its own private resources to log the error. The RAS Extension allows for a dedicated interrupt, called the critical error interrupt, to be generated by a node when such an error occurs.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{FWDD}$ For a given node, the critical error interrupt is implemented if ERRFRn.CI !=0b00.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{HJHC}$ For a given node, if the critical error interrupt is implemented, then the error recovery interrupt is also implemented.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{TCPP}$ The critical error interrupt is enabled when ERRCTRLn.CI is set to 1.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{LLPF}$ If the critical error interrupt is implemented, then when a critical error condition is recorded, the node sets ERRSTATUSn.CI to 1, regardless of whether the critical error interrupt is enabled or disabled.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{WDPT}$ If the critical error interrupt is implemented and disabled, then when a critical error condition is detected, the node records the critical error as an Uncontainable error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{XPHJ}$ Classifying the critical error condition as an Uncontainable error if the critical error interrupt is not enabled has the effect of causing the node to generate an error recovery interrupt. The node also sets ERRSTATUSn.CI to 1. If the critical error interrupt is enabled, it is IMPLEMENTATION DEFINED how the error is classified at the node. The critical error flag is set to 1 in addition to the other syndrome information for the error, which is handled in the normal way.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

See also:

Chapter B11 *Nested Vectored Interrupt Controller* on page 269.

## B15.7 RAS error records

$I_{ZFWK}$ On encountering an error, a node writes to the RAS Error records. These records can then be analyzed by software to determine whether there are any systematic problems to be dealt with.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{QDZN}$ Each node provides at least one Error record to control error reporting and expose status.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

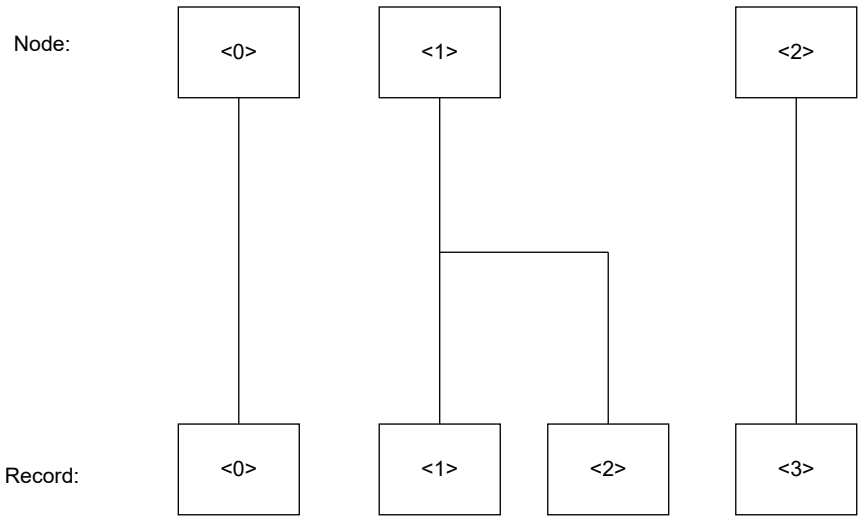
$R_{FDFQ}$ A node can provide multiple Error records if it offers multiple, logically distinct functions.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{STJR}$ If a node provides multiple Error records, these are serially indexed. Each record, other than the first record, has an ERRFRn register that is RAZ/WI and the ERRTCTRLn register is RES0.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{SDBZ}$ An example of a group containing four error records owned by three nodes is shown below:




In the diagram:

- Node<0> owns one error record: <0>.
- Node<1> owns two error records: <1> and <2>.
- Node<2> owns one error record: <3>.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JSWF}$ The standard Error records contain:

- Controls for common features, and an identification mechanism for these controls. For each node it is IMPLEMENTATION DEFINED whether the fault and error reporting mechanisms apply to both reads and writes, or whether they can be individually controlled for reads and writes.
- A status register for common status fields, such as the type and coarse characterization of the error.
- An address register, if applicable.
- IMPLEMENTATION DEFINED controls and identification registers.
- IMPLEMENTATION DEFINED status registers.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{WSCV}$ Arm recommends that the IMPLEMENTATION DEFINED status registers in the standard Error record are used for:

- Identifying a Field Replaceable Unit (FRU).
- Locating the error within the FRU.
- Optional Corrected Error counters for software to poll the rate of Corrected errors.

The architecture provides optional formats for the counters.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{QDFJ}$ The content and format of the Error records is flexible to allow implementations to select an appropriate amount of reporting.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{GVGT}$ A group of Error records can be sparsely populated, which means that not all registers in the group might contain valid information. Error record registers that are not implemented have an associated ERRFRn register field that reads as zero.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{PJGD}$ The number of Error records that can be accessed through the memory-mapped registers is IMPLEMENTATION DEFINED, and might be zero. ERRDEVID.NUM indicates the highest numbered index of the Error records that can be accessed, plus one.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JVDC}$ The content of the Error record registers is preserved over Warm reset.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{SVLS}$ Arm recommends that all Error records are remotely accessible for access by all PEs in a system, or by a Baseboard Management Controller (BMC) or System Control Processor (SCP) or debugger. The remote access mechanism is IMPLEMENTATION DEFINED but might use CoreSight-like interfaces. Arm recommends that remote access is possible when the rest of the system is in a fail state, for example when the system has locked up.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{PRWQ}$ When a new error is detected, the node:

- Sets or modifies ERRSTATUSn.{CE, DE, CI, UE, UET} to indicate the type of the new Detected error.
- The node either:
  - Overwrites the Error record with the syndrome for the new error, if it has a higher priority than the previous highest priority recorded error.
  - Keeps the syndrome for the previous error, if the new error has a lower or the same priority as the previous highest priority recorded error.
- Counts the error if it is a Corrected error and a counter is implemented.
- Sets ERRSTATUSn.V to 1.
- Generates an interrupt as required.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{XPXH}$ A counter for Corrected errors is OPTIONAL.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{LJTP}$ The overwriting of errors depends on the type of the previous highest priority error and the type of the newly recorded error. This is shown in the table below. The table uses the following abbreviations:

- **CE**: Corrected error.
- **CO**: Count and overflow. Keep the previous error syndrome and count the error. If counting the error causes an unsigned overflow of the counter set ERRSTATUSn.OF to 1.
- **CW**: Count and overwrite. Count CE if a counter is implemented and overwrite. If a counter is implemented and overflows, ERRSTATUSn.OF is set to an UNKNOWN value. Otherwise, it is IMPLEMENTATION DEFINED whether ERRSTATUSn.OF is set to 0 or unchanged.
- **CWO**: Count and overwrite or keep. The behavior is IMPLEMENTATION DEFINED and described by the value of ERRFRn.CEO:

  **0**: Count CE if a counter is implemented and keep the previous error syndrome.

  **1**: Count CE. If ERRSTATUSn.OF == 1 before the CE is counted, keep the previous syndrome. Otherwise record the new error syndrome.

If the counter overflows or if no counter is implemented ERRSTATUSn.OF is set to 1.

- **DE**: Detected error.
- **O**: Overflow. Keep the previous error syndrome and set ERRSTATUSn.OF to 1.
- **UEO**: Restartable error.
- **UER**: Recoverable error.
- **UEU**: Unrecoverable error.
- **UC**: Uncontainable error.
- **WO**: Overwrite and overflow. ERRSTATUSn.OF is set to 1.

If no counter is implemented, **CW** behaves the same as **W**, and **CWO** and **CO** behave the same as **O**.

| **Previous error type** | **New detected error type** | | | | | |
|---|---|---|---|---|---|---|
| | CE | DE | UEO | UER | UEU | UC |
| | CW | W | W | W | W | W |
| CE | CWO | WO | WO | WO | WO | WO |
| DE | CO | O | WO | WO | WO | WO |
| UEO | CO | O | O | WO | WO | WO |
| UER | CO | O | O | O | WO | WO |
| UEU | CO | O | O | O | O | WO |
| UC | CO | O | O | O | O | O |

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{GPFK}$ When a node generates an interrupt or exception, it ensures that any subsequent reads to the Error records return the updated values.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{ZCKJ}$ ERRGSRn contains a read-only copy of ERRSTATUSn.V.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.8 Multiple BusFault exceptions

$I_{RDHF}$ Asynchronous BusFaults can be generated by multiple exception conditions. The architecture does not define relative priorities.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{SCCP}$ It is IMPLEMENTATION DEFINED whether bus errors that were generated by multiple exception conditions are taken as a single BusFault exception.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{SWRS}$ On taking a BusFault exception, whether for one or more BusFault exception conditions, the effects of the BusFault exception or exceptions on the state of the PE is reported in RFSR.UET as any one of the following:

- Uncontained error.
- Unrecoverable error.
- Restartable error.
- Recoverable error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$R_{JJCW}$ An Error Synchronization Barrier operation requires that all Unrecoverable errors are synchronized. If there are multiple requests outstanding, they are all synchronized by a single Error Synchronization Barrier operation.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

## B15.9 Minimal RAS implementation

$I_{QJVW}$ The minimal RAS Extension is mandatory in an Armv8.1-M implementation. All additional RAS features are optional.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{NPMM}$ The RAS Extension is designed to provide a low implementation cost for devices that have limited RAS support.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{DHMH}$ The minimal implementation of the RAS Extension is:

- The `ESB` instruction. This might be implemented as a NOP for implementations that do not offer RAS-induced BusFaults.
- RFSR. This might be implemented as RAZ/WI.
- ERRDEVID. This might be implemented as RAZ/WI.
- AIRCR.IESB This might be implemented as RAZ/WI.
- ID_PFR0.RAS reads as a `0b0010`.

This provides the necessary architectural support while offering no actual RAS reporting functionality.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

$I_{GRQK}$ Error record registers that are not implemented are RES0. In a minimal implementation, there might be no Error records.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **RAS**.*

# Part C
## Armv8-M Instruction Set

# Chapter C1
# Instruction Set Overview

This chapter provides a definition of the *instruction descriptions* contained in Chapter C2 *Instruction Specification* on page 444. It contains the following sections:

## C1.1 Instruction set

$R_{NPFK}$ There is one instruction set, called T32.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

C1.4 *Instruction set encoding information* on page 435.

Chapter C2 *Instruction Specification* on page 444.

# C1.2 Format of instruction descriptions

$I_{XQQV}$ Each instruction description in Chapter C2 *Instruction Specification* on page 444 has the following content:

1. A title.
2. A short description.
3. The instruction encoding or encodings.
4. Any alias conditions, if applicable.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.2.1 The title

$I_{RFFL}$ The title of an instruction description includes the base mnemonic or mnemonics for the instruction. This is part of the assembler syntax, for example SUB.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BSWN}$ If different forms of an instruction use the same base mnemonic, each form has its own description. In this case, the title is the mnemonic followed by a short description of the instruction form in parentheses. This is most often used when an operand is an immediate value in one instruction form, but is a register in another form.

For example, in Chapter C2 *Instruction Specification* on page 444 the Armv8-M Instruction Set there are the following titles for different forms of the ADD instruction:

- `ADD (SP plus immediate)`
- `ADD (SP plus register)`
- `ADD (immediate)`
- `ADD (immediate to PC)`
- `ADD (register)`

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{RKXC}$ Where an instruction has more than one variant, the descriptions might be combined, for example for `CDP` and `CDP2`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.2.2 A short description

$I_{QNXW}$ This briefly describes the function of the instruction. The short description is not a complete description of the instruction and must be read in conjunction with the instruction encoding, mnemonic, alias conditions, assembler symbols, pseudocode and any applicable notes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.2.3 The instruction encoding or encodings

$R_{LTJB}$ Instruction descriptions in this manual contain:

- An encoding section, containing one or more encoding diagrams, each followed by some decode pseudocode that:
  1. Picks out any encoding-specific special cases.

2. Translates the fields of the encoding into inputs for the common pseudocode of the instruction

- An operation section, containing common pseudocode that applies to all of the encodings being described. The Operation section pseudocode contains a call to the `EncodingSpecificOperations()` function which triggers the decode pseudocode, either at its start or only after a Condition code check performed by **`if`** `ConditionPassed()` **`then`**.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BDDV}$ An encoding diagram specifies each bit of the instruction encoding as one of the following:

- A mandatory 0 or 1, represented in the diagram as 0 or 1. If the PE attempts to decode and execute the instruction and a bit does not have a mandatory value, the encoding corresponds to a different instruction.
- A *should be* 0 or *should be* 1, represented in the diagram as (0) or (1). If the PE attempts to decode and execute the instruction and a bit does not have the *should be* value, the instruction is CONSTRAINED UNPREDICTABLE.
- A named single bit or a bit in a named multi-bit field.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BBZT}$ An encoding diagram matches an instruction if all mandatory bits are identical in the encoding diagram and the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DKWV}$ Between each encoding diagram and its T <*n*> heading, there is an italicized statement that describes which *Armv8-M variant* the encoding is present in. For example, *Armv8-M Main Extension only.*

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JSBT}$ The instruction description shows the instruction encoding diagram, or, if the instruction has multiple encodings, shows all of the encoding diagrams. The heading for each encoding is the letter *T* followed by an arbitrary number, usually between 1 and 5.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{FQDP}$ Below each encoding diagram is the *assembler syntax prototype* for that encoding, written in typewriter font. The assembler syntax prototype describes the syntax that can be used in the assembler to select this encoding, and also the syntax that is used when disassembling this encoding.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BLJR}$ In some cases an encoding has multiple variants of *assembler syntax prototype*, when the prototype differs depending on the value in one or more of the encoding fields. In these cases, the correct variant to use can be identified by either:

- Its subheading.
- An annotation to the syntax.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B6.3 *Endianness* on page 188.

C1.2.6 *Pseudocode describing how the instruction operates* on page 425.

### C1.2.4 Any alias conditions, if applicable

$I_{BMHC}$ Alias conditions are an optional part of an instruction description. If included, it describes the set of conditions for which an alternative mnemonic and its associated assembler syntax prototypes are preferred for disassembly by a disassembler. It includes a link to the alias instruction description that defines the alternative syntax. The alias syntax and the original syntax can be used interchangeably in the assembler source code.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{RBCM}$ Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{ZJKQ}$ Arm recommends that where possible, the alias is used.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.2.5 Standard assembler syntax fields

$I_{RHCC}$ This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{LBNB}$ UAL describes the syntax for the mnemonic and the operands of each instruction. Operands can also be referred to as *Assembler symbols*. In addition, UAL assumes that instructions and data items can be given labels. It does not specify the syntax to be used for labels, see the assembler documentation for these details.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{DPLM}$ The *Assembler symbols* subsection of an instruction description contains a list of the symbols that the assembler syntax prototype or prototypes use.

The following conventions are used:

`< >`: Angle brackets. Any symbol enclosed by these is mandatory. For each symbol, there is a description of what the symbol represents. The description usually also specifies which encoding field or fields encodes the symbol.

`{ }`: Brace brackets. Any symbol enclosed by these is optional. For each optional symbol, there is a description of what the symbol represents and how its presence or absence is encoded.

In some assembler syntax prototypes, some brace brackets are mandatory, for example if they surround a register list. When the use of brace brackets is mandatory, they are separated from other syntax items by one or more spaces.

`#` : Usually precedes a numeric constant. All uses of # are optional in assembler source code. Arm recommends that disassemblers output the # where the assembler syntax prototype includes it.

`+/-`: Indicates an optional `+` or `–` sign. If neither is coded, `+` is assumed.

`!` : Indicates that the result address is written back to the base register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MBQS}$ Single spaces are used for clarity, to separate syntax items. Where a space is mandatory, the assembler syntax prototype shows two or more consecutive spaces.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{SXWN}$ Any characters not shown in this conventions list must be coded exactly as shown in the assembler syntax prototype. Apart from brace brackets, these characters are used as part of a meta-language to define the architectural assembler syntax prototype for an instruction encoding, but have no architecturally defined significance in the input to an assembler or in the output from a disassembler.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QZDB}$ UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality. The following assembler syntax prototype fields are standard across all or most instructions:

`<c>`: Specifies the condition under which the instruction is executed. If `<c>` is omitted, it defaults to *always* (`AL`).

`<q>`: Specifies one of the following optional assembler qualifiers on the instruction:

`.N`

Meaning narrow. The assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

`.W`

Meaning wide. The assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.

If neither `.W` nor `.N` is specified, the assembler can select either a 16-bit or 32-bit encoding. If both encoding lengths are available, it must select a 16-bit encoding. In the few cases where more than one encoding of the same length is available for an instruction, the rules for selecting between them are instruction-specific and are part of the instruction description.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{HGFS}$ The following assembler syntax prototype field is standard across MVE instructions subject to VPT predictation:

`<v>`: Specifies a VPT predication block. This field is only available in implementations that include MVE, and it is only available inside a VPT block. Inside a VPT block, `<v>` can have one of the following values:

`T`

Indicates that the instruction is in the THEN section of a VPT block.

`E`

Indicates that the instruction is in the ELSE section of a VPT block.

`<v>` does not affect the encoding of the instruction, and only highlights to the programmer that some of the vector lanes might be masked because of VPT predication. The use of `T` or `E` outside a VPT block produces an assembler error.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

$I_{BWNR}$ 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B5.6.2 *VPT predication* on page 177.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

### C1.2.6 Pseudocode describing how the instruction operates

$I_{RTDZ}$ Each instruction description includes pseudocode that provides a precise description of what the instruction does.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{LRFZ}$ In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{NLPM}$ Where the pseudocode describes UNPREDICTABLE behavior the constraints on that behavior are described in the Operation section.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BNVW}$ Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CRWM}$ Pseudocode describes the exact rules when an UNDEFINED instruction fails its Condition code check.

In such cases, the `UNDEFINED` pseudocode statement lies inside the if `ConditionPassed()` then ... structure, either directly or in the `EncodingSpecificOperations()` function call, and so the pseudocode indicates that the instruction executes as a `NOP`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{MZKZ}$ Pseudocode does not describe the exact ordering requirements when a single floating-point instruction generates more than one floating-point exception and one or more of those floating-point exceptions is trapped.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{JMFG}$ An exception can be taken during execution of the pseudocode for an instruction, either explicitly as a result of the execution of a pseudocode function, or implicitly, for example if an interrupt is taken during execution of an `LDM` instruction. If this happens, the pseudocode does not describe the extent to which the normal behavior of the instruction occurs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

Chapter E1 *Arm Pseudocode Definition* on page 1752.

B6.10 *Ordering requirements for memory accesses* on page 200.

E1.1.1 *General limitations of Arm pseudocode* on page 1753.

C1.3.3 *Conditional execution of undefined instructions* on page 430.

B4.12 *Priority of Floating-point exceptions relative to other Floating-point exceptions* on page 165.

B3.18 *Exception handling* on page 101.

B3.22 *Exception return* on page 115.

### C1.2.7 Use of labels in UAL instruction syntax

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

1. Calculate the `PC` or `Align(PC,4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a T32 instruction. The `Align(PC,4)` value of an instruction is its `PC` value ANDed with `0xFFFFFFFC` to force it to be word-aligned.
2. Calculate the offset from the `PC` or `Align(PC,4)` value of the instruction to the address of the labeled instruction or literal data item.
3. Assemble a *PC-relative* encoding of the instruction, that is, one that reads its `PC` or `Align(PC,4)` value and

adds the calculated offset to form the required address.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{TCVF}$ 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DLVP}$ The following instructions include a label:

- B and BL.
- CBNZ and CBZ.
- LDC, LDC2, LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH, PLD, PLI, and VLDR:
  - When the assembler calculates an offset of 0 for the normal syntax of these instructions, it must assemble an encoding that adds 0 to the `Align(PC,4)` value of the instruction. Encodings that subtract 0 from the `Align(PC,4)` value cannot be specified by the normal syntax.
  - There is an alternative syntax for these instructions that specifies the addition or subtraction and the immediate offset explicitly. In this syntax, the label is replaced by `[PC, #+/-<imm>]`, where:
    - **+/-**: Is + or omitted to specify that the immediate offset is to be added to the `Align(PC,4)` value, or - if it is to be subtracted.
    - **<imm>**: Is the immediate offset.
  - This alternative syntax makes it possible to assemble the encodings that subtract 0 from the `Align(PC,4)` value, and to disassemble them to a syntax that can be re-assembled correctly.
- ADR:
  - When the assembler calculates an offset of 0 for the normal syntax of this instruction, it must assemble the encoding that adds 0 to the `Align(PC,4)` value of the instruction. The encoding that subtracts 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KFSF}$ From Armv8.1-M the following instructions also include a label:

- BF, BFX, BFL, BFLX, BFCSEL.
- LE and LETP.
- WLS and WLSTP.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

### C1.2.8 Using syntax information

$I_{BJGX}$ For a particular encoding:

- There is usually more than one assembler syntax prototype variant that assembles to it.
- The exact set of prototype variants that assemble to it usually depends on the operands to the instruction, for example the register numbers or immediate constants. As an example, for the AND (register) instruction, the syntax AND R0, R0, R8 selects a 32-bit encoding, but AND R0, R0, R1 selects a 16-bit encoding.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

`I_HQSS` For each instruction encoding that belongs to a target instruction set, an assembler can use the information in the encoding to determine whether it can use that particular encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecturally preferred encoding is the 16-bit encoding. This means that the assembler must use the 16-bit encoding instead of the 32-bit encoding.
- If multiple encodings of the same width can encode the instruction, the assembler syntax indicates the preferred encoding, and how software can select other encodings if required. Each encoding also documents UAL syntax that selects it in preference to any other encoding. If no encodings of the target instruction set can encode the instruction requested by the UAL source, the assembler normally generates an error that indicates that the instruction is not available in that instruction set.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

# C1.3 Conditional execution

$I_{XDMQ}$ *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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{SPPQ}$ Most T32 instructions are unconditional. Conditional execution in T32 code can be achieved using any of the following instructions:

- A 16-bit conditional branch instruction, with a branch range of -256 to +254 bytes. See B for details.
- A 32-bit conditional branch instruction, with a branch range of approximately ± 1MB. See B for details.
- 16-bit Compare and Branch on Zero and Compare and Branch on Nonzero instructions, with a branch range of +4 to +130 bytes. See CBNZ, CBZ for details.
- A 16-bit If-Then instruction that makes up to four following instructions conditional. See IT for details. The instructions that are made conditional by an IT instruction are called its *IT block*. Instructions in an IT block must either all have the same condition, or some can have one condition, and others can have the inverse condition.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FNBQ}$ In T32 instructions, the condition (if it is not AL) is encoded in a preceding IT instruction, other than B, CBNZ and CBZ. Some conditional branch instructions do not require a preceding IT instruction, and include a condition code in their encoding.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{BDMC}$ The following table shows the conditions that are available for conditionally executed instructions.

| cond | Mnemonic extension | Meaning, integer artihmetic | Meaning, Floating-point arithmetic | APSR condition flags |
|---|---|---|---|---|
| 0000 | EQ | Equal | Equal | Z == 1 |
| 0001 | NE | Not equal | Not equal, or unordered | Z == 0 |
| 0010 | CS | Carry set | Greater than, equal or unordered | C == 1 |
| 0011 | CC | Carry clear | Less than | C == 0 |
| 0100 | MI | Minus, negative | Less than | N == 1 |
| 0101 | PL | Plus, positive or zero | Greater than, equal or unordered | N == 0 |
| 0110 | VS | Overflow | Unordered | V == 1 |
| 0111 | VC | No overflow | Not unordered | V == 0 |
| 1000 | HI | Unsigned higher | Greater than or unordered | C == 1 and Z == 0 |
| 1001 | LS | Unsigned lower or same | Less than or equal | C == 0 or Z == 1 |
| 1010 | GE | Signed greater than or equal | Greater than or equal | N == V |
| 1011 | LT | Signed less than | Less than or unordered | N != V |
| 1100 | GT | Signed greater than | Greater than | Z == 0 and N == V |
| 1101 | LE | Signed less than or equal | Less than, equal or unordered | Z == 1 and N != V |
| 1110 | None (AL) | Always (unconditional) | Always (unconditional) | Any |

Unordered means at least one NaN operand.

HS (unsigned higher or same) is a synonym for CS.

LO (unsigned lower) is a synonym for CC.

AL is an optional mnemonic extension for always, except in IT instructions. See IT for details.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.3.1 Conditional instructions

$R_{WRJS}$ The instructions that are made conditional by an `IT` instruction must be written with a condition after the mnemonic. These conditions must match the conditions imposed by the IT instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{WVXC}$ An example of $R_{WRJS}$ is:

```
1       ITTEE EQ
2       ADDEQ R0, R1
3       SUBEQ R2, R3
4       ADDNE R4, R5
5       SUBNE R6, R7
```

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{THGJ}$ Some instructions cannot be made conditional by an IT instruction. Some instructions can be conditional if they are the last instruction in the IT block, but not otherwise, see the individual instruction descriptions for details.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TGXF}$ 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also

- `IT` instruction

### C1.3.2 Pseudocode details of conditional execution

$R_{NMVJ}$ The `CurrentCond()` pseudocode function prototype returns a 4-bit condition specifier as follows:

- For the T1 and T3 encodings of the Branch instruction, it returns the 4-bit `cond` field of the encoding.
- For all other T32 instructions:
  - If `ITSTATE.IT<3:0>` != `'0000'` it returns `ITSTATE.IT<7:4>`
  - If `ITSTATE.IT<7:0>` == `'00000000'` it returns `'1110'`
  - Otherwise, execution of the instruction is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LTPQ}$ The `ConditionPassed()` function calls the `ConditionHolds()` function to determine whether the instruction must be executed.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also

C1.3.5 *ITSTATE* on page 431.

`B`.

### C1.3.3 Conditional execution of undefined instructions

$R_{NPNF}$ The conditional execution applies to all instructions. This includes undefined instructions and other instructions that would cause entry to the UsageFault or the UNDEFINSTR UsageFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PCJZ}$ If such an instruction fails its condition code check the instruction behaves as a NOP and does not cause an UsageFault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.3.4 Interaction of undefined instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

$R_{NZWQ}$ If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED, then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED, then the instruction is CONSTRAINED UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.3.5 ITSTATE

$I_{RGFT}$ ITSTATE is held in EPSR.IT.

This register holds the If-Then Execution state bits for the T32 IT instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{QKPG}$ EPSR.IT and ITSTATE divide into two subfields:

IT[7:5]

Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the IT instruction.

This subfield is 0b000 when no IT block is active.

IT[4:0]

Encodes:

* The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is indicated by the position of the least significant 1 in this field which is bit [4-size of the block].

* The value of the least significant bit, bit[0], of the condition code for each instruction in the block.

* Changing the value of the least significant bit of a condition code from 0 to 1 inverts the condition code. For example cond 0000 is EQ, and cond 0001 is NE.

This subfield is 0b00000 when no IT block is active.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{FPNH}$ When an IT instruction is executed, IT bits[7:0] are set according to the condition in the instruction, and the *Then* and *Else* (Tand E) parameters in the instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XKGZ}$ An instruction in an IT block is conditional. The condition used is the current value of IT[7:4]. When an instruction in an IT block completes its execution normally, ITSTATE is advanced by shifting IT bits[4:0] left by 1 bit.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{VQJM}$ For example:

```
                    IT[7:5]   IT[4:0]
ITTEE EQ            000       00111
ADDEQ R0, R1        000       01110
SUBEQ R2, R3        000       11100
ADDNE R4, R5        000       11000
SUBNE R6, R7        000       00000
```

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{KQBQ}$ 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{FJLN}$ In the following table, *P* represents the base condition or the inverse of the base condition.

| IT Bits | | | | | | |
|---|---|---|---|---|---|---|
| [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 |

Combinations of the IT bits not shown in this table are reserved.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.3.6 Pseudocode details of ITSTATE operation

$I_{JLKP}$ `ITAdvance()` describes how ITSTATE advances after normal execution.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{ZGZL}$ `InITBlock()` and `LastInITBlock()` test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.3.7 SVC and ISTATE

$R_{TSWQ}$ The `ReturnAddress()` for an `SVC` instruction must point to the instruction after the `SVC` instruction and advance ITSTATE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{QWZW}$ When an `SVC` instruction is escalated to HardFault resulting in lockup the `ReturnAddress()` is `0xEFFFFFFE`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

See also:

B3.33 *Lockup* on page 139.

## C1.3.8 CONSTRAINED UNPREDICTABLE behavior and IT blocks

$R_{WWVX}$ Branching into an IT block, other than by way of exception return or exit from Debug state, leads to CONSTRAINED UNPREDICTABLE behavior. Execution starts from the address that is determined by the branch, but each instruction

in the IT block is:

- Executed as if the instruction is not in an IT block, meaning that the instruction is executed unconditionally.
- Executed as if the instruction had passed its Condition code check within an IT block.
- Executed as a NOP. That is, the instruction behaves as if it had failed the Condition code check.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CPDC}$ For exception returns or Debug state exits that cause EPSR.IT to be set to a reserved value with a nonzero value in EPSR.IT, the EPSR.IT bits are forced to `0b00000000`.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, Debug state requires Halting debug.*

$R_{HVNS}$ Exception returns or Debug state exits that set EPSR.IT to a non-reserved value can occur when the flow of execution returns to a point:

- Outside an IT block, but with the EPSR.IT bits set to a value other than `0b00000000`.
- Inside an IT block, but with a different value of the EPSR.IT bits than if the IT block had been executed without an exception return or Debug state exit.

In this case the instructions at the target of the exception return or Debug state exit does one of the following:

- Execute as if they passed the Condition code check for the remaining iterations of the EPSR.IT state machine.
- Execute as NOPs. That is, they behave as if they failed the Condition code check for the remaining iterations of the EPSR.IT state machine.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, Debug state requires Halting debug.*

$R_{LLDK}$ A number of instructions in the architecture are described as being CONSTRAINED UNPREDICTABLE either:

- Anywhere within an IT block.
- As an instruction within an IT block, other than the last instruction within an IT block.

Unless otherwise stated in this reference manual, when these instructions are committed for execution, one of the following occurs:

- An UNDEFINED exception is taken.
- The instructions are executed as if they had passed the condition code check.
- The instructions execute as NOPs, as if they had failed the condition code check.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{NJKF}$ The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BWMN}$ Branch instructions or other non-sequential instructions that change the PC are CONSTRAINED UNPREDICTABLE in an IT block. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the EPSR.IT state machine is treated in one of the following ways:

- EPSR.IT is cleared to 0.
- EPSR.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{XHBL}$ This behavior does not apply to an instruction that is the last instruction in an IT block.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TMWN}$ The instructions that are addressed by the updated PC does one of the following:

- Execute as if they had passed the condition code check for the remaining iterations of the EPSR.IT state machine.
- Execute as NOPs. That is, they behave as if they had failed the condition code check for the remaining iterations of the EPSR.IT state machine.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KVXD}$ The remaining iterations of the EPSR.IT state machine behave in one of the following ways:

- The EPSR.IT state machine advances as if it were in an IT block.
- The EPSR.IT bits are ignored.
- The EPSR.IT bits are forced to `0b00000000`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GZBX}$ Execution of an instruction inside an IT block with ITSTATE set to zero, an ICI value, or a value that is inconsistent with the IT block is UNPREDICTABLE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VBCG}$ In the `VIWDUP` and `VDWDUP` instructions the following conditions result in CONSTRAINED UNPREDICTABLE behavior:

- Rn is not a multiple of imm.
- Rm is not a multiple of imm.
- Rn >= Rm.

The CONSTRAINED UNPREDICTABLE behavior is that the resulting values of Rn and Qd become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards. The extension requirements are - **MVE**.*

See also:

B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

B3.5.2 *Execution Program Status Register (EPSR)* on page 74.

# C1.4 Instruction set encoding information

## C1.4.1 UNDEFINED and UNPREDICTABLE instruction set space

$I_{FLRZ}$ An attempt to execute an unallocated instruction results in either:

- UNPREDICTABLE behavior. The instruction is described as UNPREDICTABLE.
- An UNDEFINSTR UsageFault. The instruction is described as UNDEFINED.
- Unallocated instructions in the NOP hint space behave as NOPs.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{KDXB}$ An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XDBQ}$ 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TRHK}$ Unless otherwise specified, a T32 instruction that is provided by one or more of the architecture extensions is either UNPREDICTABLE or UNDEFINED in an implementation that does not include those extensions. See the individual instruction descriptions for details.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.4.2 Pseudocode descriptions of operations on general-purpose registers and the PC

$R_{HRGP}$ In pseudocode, the uses of the R[] function are:

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

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CHTM}$ The use of `RSPCheck()` returns the value of the current SP

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GXHR}$ The function `Rz` returns zeroes if the PC is called.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

See also:

`R[].RSPCheck()`

`Rz`

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

## C1.4.3 Use of 0b1111 as a register specifier

$R_{WMVJ}$ All use of the PC as a named register specifier for a source register that is described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual does one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to be executed as a NOP.
- Read an UNKNOWN value for the source register that is specified as the PC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{BGJG}$ All use of the PC as a named register specifier for a destination register that is described as CONSTRAINED UNPREDICTABLE in the pseudocode or in other places in this reference manual does one of the following:

- Cause the instruction to be treated as UNDEFINED.
- Cause the instruction to be executed as a NOP.
- Ignore the write.
- Branch to an UNKNOWN location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{QVWL}$ The choice between the behavior of the PC as a source or destination register might vary in some implementations from instruction to instruction, or between different instances of the same instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{LXPR}$ For instructions that specify two destination registers and if one is specified as the PC, then the other destination register of the pair is UNKNOWN. The CONSTRAINED UNPREDICTABLE behavior for the write to the PC is either to ignore the write or to branch to an UNKNOWN location.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DRSS}$ An instruction that specifies the PC as a Base register and specifies a base register writeback is CONSTRAINED UNPREDICTABLE and behaves as if the PC is both the source and destination register.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XLVX}$ For instructions that affect any or all of APSR.{N, Z, C, V} or APSR.GE when the register specifier is not the PC, any flags that are affected by an instruction that is CONSTRAINED UNPREDICTABLE become UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JFGT}$ For MRC instructions that use the PC as the destination register descriptor (and therefore target APSR.{N, Z, C, V}) and where these instructions are described as being CONSTRAINED UNPREDICTABLE the status of the flags becomes UNKNOWN.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{XPBT}$ Multi-access instructions that load the PC from Device memory are CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:

- The instruction loads the PC from the memory location as if the memory location had the Normal Non-cacheable attribute.
- The instruction generates a MemManage fault.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{RTKM}$ All unallocated or reserved values of fields with allocated values within the memory-mapped registers that are described in this reference manual behave, unless otherwise stated in the register description, in one of the following ways:

- The encoding maps onto any of the allocated values, but otherwise does not cause CONSTRAINED UNPREDICTABLE behavior.
- The encoding causes effects that could be achieved by a combination of more than one of the allocated encodings.

- The encoding causes the field to have no functional effect.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{QXNP}$ When a value of `0b1111` is permitted as a register specifier, as indicated in the individual instruction descriptions, a variety of meanings is possible. For register reads, these meanings are:

- 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>`.)
- Read the word-aligned PC value, that is, the address of the current instruction + 4, with bits [1:0] forced to zero. The base register of `LDC`, `LDR`, `LDRB`, `LDRD` (pre-indexed, no write-back), `LDRH`, `LDRSB`, and `LDRSH` instructions can be the word-aligned PC. This enables PC-relative data addressing. In addition, some encodings of the `ADD` and `SUB` instructions permit their source registers to be `0b1111` for the same purpose.
- Read zero. This is done in some cases when one instruction is a special case of another, more general instruction, but with one operand zero. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. An example of this is the descriptions of `MOV (register)` and `ORR (register)`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{KVHQ}$ When a value of `0b1111` is permitted as a register specifier, as indicated in the individual instruction descriptions, a variety of meanings is possible. For register writes, these meanings are:

- The PC can be specified as the destination register of an `LDR` instruction. This is done by encoding Rt as `0b1111`. The loaded value is treated as an address, and the effect of execution is a branch to that address. bit[0] of the loaded value selects the Execution state after the branch and must have the value 1.
- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. An example of this is the descriptions of `TST (register)` and `AND (register)`.
- If the destination register specifier of an `LDRB`, `LDRH`, `LDRSB`, or `LDRSH` instruction is `0b1111`, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an `MRC` instruction is `0b1111`, bits[31:28] of the value transferred from the coprocessor are written to the N, Z, C, and V flags in the APSR, and bits[27:0] are discarded.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.4.4 Use of 0b1101 as a register specifier

#### SP[1:0] definition

$R_{DSDB}$ Bits [1:0] of SP must be 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.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

#### 32-bit T32 instruction support for SP

$R_{SKNR}$ Use of the SP in T32 instructions and 16-bit data processing instructions is restricted to the following cases:

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

```
1         MOV     SP,Rm
2         MOV     Rn,SP
```

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

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

- SP as a base register, Rn, of any load or store instruction. This supports SP-based addressing for load, store, or memory hint instructions, with positive or negative offsets, with and without write-back.
- SP as the first operand, Rn, in any `ADD{S}`, `CMN`, `CMP`, or `SUB{S}` instruction. The add and subtract instructions support SP-based address generation, with the address going into a general-purpose register. `CMN` and `CMP` can check the stack pointer.
- SP as the transferred register, Rt, in any `LDR` or `STR` instruction.
- SP as the address in a `POP` or `PUSH` instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{MRNT}$ Where an instruction states that the SP is UNPREDICTABLE and SP is used:

- The value that is read or written from or to the SP is UNKNOWN.
- The instruction is permitted to be treated as UNDEFINED.
- If the SP is being written, it is UNKNOWN whether a stack-limit check is applied.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.4.5 16-bit T32 instruction support for SP

$R_{STHZ}$ Arm deprecates any other use of the SP in T16 instuctions. This affects the high register forms of `CMP` and `ADD`, where Arm deprecates the use of SP as `Rm`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.4.6 Branching

$I_{PVGL}$ Writing an address to the PC causes either a simple branch to that address or an *interworking* branch.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CCVD}$ A simple branch is performed by `BranchWritePC()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TQSZ}$ A simple branch is performed by `BranchTo()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{XMGH}$ An interworking branch is performed by `BXWritePC()`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{JTJJ}$ An interworking branch is performed by `BranchReturn()`.

*Applies to an implementation of the architecture from **Armv8.1-M** onwards.*

$R_{CWSL}$ Branching can occur in cases where `0b1111` is not a register specifier. In these cases, instructions write the PC either:

- Implicitly, for example, `b<cond>`.
- By using a register mask rather than a register specifier, for example `LDM`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$I_{FLZZ}$ The address to branch to can be:

- A loaded value, for example `LDM`.
- A register value, for example `BX`.
- The result of a calculation, for example `TBB` or `TBH`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WQBX}$ The following table summarizes the branch instructions in the T32 instruction set.

| Instruction | See | Range, T32 |
|---|---|---|
| Branch to target address | `B` | ±16MB |
| Compare and Branch on Nonzero,<br>Compare and Branch on Zero | `CBNZ, CBZ` | 0-126 bytes |
| Call a subroutine | `BL` | ±16MB |
| Call a subroutine, optionally change Security state | `BLX, BLXNS` | Any |
| Branch to target address, change to Non-secure state | `BX, BXNS` | Any |
| Table Branch (byte offsets)<br>Table Branch (halfword offsets) | `TBB, TBH` | 0-510 bytes<br>0-31070 bytes |

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{GJML}$ Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TPTF}$ A load instruction that targets the PC behaves as a branch instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

### C1.4.7 Instruction set, interworking and interstating support

$R_{LBQC}$ The following instructions are Interworking branches:

- `BX` and `BLX`.
- `POP (mulitple registers)` and all forms of `LDM`, when the register list includes the PC.
- `LDR (immediate)`, `LDR (literal)`, and `LDR (register)`, with equal to the PC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{WRZR}$ The value of bit[0] of an interworking branch instruction is not stored in the PC. Bit[0] of an interworking branch instruction sets EPSR.T. If EPSR.T is cleared to 0 an INVSTATE UsageFault or HardFault is generated on the next instruction the PE attempts to execute.

*Applies to an implementation of the architecture from **Armv8.0-M**. Note, requires M for INVSTATE UsageFault.*

$R_{GLPL}$ The following instructions are *interstating branches*:

- `BXNS` and `BLXNS`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{GJMJ}$ When an interstating branch is executed in Secure state, bit[0] of the target address indicates the target Security state:

**0**: The target Security state is Non-secure state.

**1**: The target Security state is Secure state.

The value of bit[0] of an interstating branch instruction is not stored in the PC.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

$R_{WNSX}$ Interstating branches are UNDEFINED when executing in Non-secure state.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **S**.*

See also:

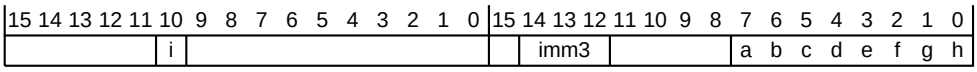
C1.1 *Instruction set* on page 421.

`BXWritePC()`.

B3.15 *Security state transitions* on page 96.

# C1.5 Modified immediate constants

$R_{JVCL}$ The encoding of modified immediate constants in T32 instructions is:




*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{TCLZ}$ The table shows the range of modified immediate constants available in T32 data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

| i:imm3:a | <const> | Carry flag set |
|---|---|---|
| 0000x | 00000000 00000000 00000000 abcdefgh | No |
| 0001x | 00000000 abcdefgh 00000000 abcdefgh | No |
| 0010x | abcdefgh 00000000 abcdefgh 00000000 | No |
| 0011x | abcdefgh abcdefgh abcdefgh abcdefgh | No |
| 01000 | 1bcdefgh 00000000 00000000 00000000 | Yes, to 1 |
| 01001 | 01bcdefg h0000000 00000000 00000000 | Yes, to 0 |
| 01010 | 001bcdef gh000000 00000000 00000000 | Yes, to 0 |
| 01011 | 0001bcde fgh00000 00000000 00000000 | Yes, to 0 |
| - | 8-bit values shifted to other positions | Yes, to 0 |
| - | | |
| - | | |
| 11101 | 00000000 00000000 000001bc defgh000 | Yes, to 0 |
| 11110 | 00000000 00000000 0000001b cdefgh00 | Yes, to 0 |
| 11111 | 00000000 00000000 00000001 bcdefgh0 | Yes, to 0 |

This table shows the immediate constant value in binary form, to relate `abcdefgh` to the encoding diagram. In assembly syntax, the immediate value is specified as a decimal integer by default.

The setting of the Carry flag will only apply if a logical oeperation with a modified immediate constant can set the flags.

Where i:imm3:a is `0001x`, `0010x` or `0011x` the instruction will be UNPREDICTABLE if `abcdefgh == 0b00000000`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.5.1 Operation of modified immediate constants

$R_{TLFG}$ `T32ExpandImm()` and `T32T32ExpandImm_C()` describe the operation of modified immediate constants.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{PHBG}$ The operation of modified immediate constants are UNPREDICTABLE where both:

- `hw2`[7:0] == `0b0000000`.
- `hw1`[10] == `0` and either:
  - `hw2` [14:12] == `0b001`.
  - `hw2` [14:12] == `0b010`.
  - `hw2` [14:12] == `0b011`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

## C1.6 NOP-compatible hint instructions

$I_{BJRT}$ A hint instruction only provides an indication to the PE. It is not required that the PE perform an operation on a hint instruction.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{VXQV}$ A NOP-compatible hint instruction either:

- Acts as a NOP (No Operation) instruction.
- Performs some IMPLEMENTATION DEFINED behavior.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{DBNQ}$ A PE without the Main Extension only supports the 16-bit encodings of the Armv8-M NOP-compatible hint instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **!M**.*

$R_{DJQL}$ A PE with the Main Extension supports both the 16-bit and the 32-bit encodings of the Armv8-M NOP-compatible hint instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **M**.*

See also

Hints, T16.

Hints, T32.

## C1.7 SBZ or SBO fields in instructions

$I_{PWBN}$ Many of the instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

$R_{CKJK}$ If the instruction bit pattern of an instruction is executed with these fields not having the *should-be* values, one of the following must occur:

- The instruction is UNDEFINED.
- The instruction executes as a NOP.
- The instruction operates as if the bit had the *should-be* value.
- Any destination registers of the instruction become UNKNOWN.

The exceptions to this rule are:

- `LDM, LDMIA, LDMFD`.
- `LDMDB, LDMEA`.
- `LDR (immediate)`.
- `LDRB (immediate)`.
- `LDRD (immediate)`.
- `LDRH (immediate)`.
- `LDRSB (literal)`.
- `LDRSH (literal)`.
- `POP (multiple registers)`.
- `PUSH (multiple registers)`.
- `SDIV`.
- `STM, STMIA, STMEA`.
- `STMDB, STMFD`.
- `UDIV`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

# Chapter C2
# Instruction Specification

This chapter specifies the Armv8-M instruction set. It contains the following sections:

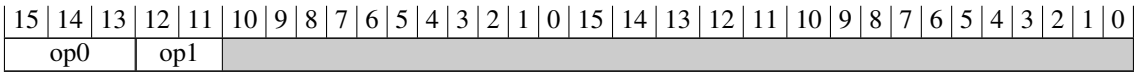
# C2.1 Top level T32 instruction set encoding

The T32 instruction stream is a sequence of halfword-aligned halfwords. Each T32 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 the value of bits[15:11] of the halfword being decoded is one of the following, the halfword is the first halfword of a 32-bit instruction:

- 0b11101.
- 0b11110.
- 0b11111.

Otherwise, the halfword is a 16-bit instruction.




**This table shows the decode field values and the associated subgroups:**

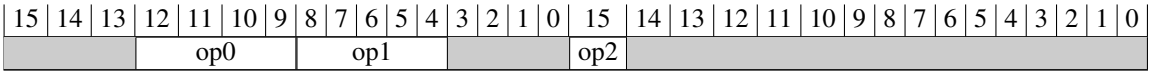
| op0 | op1 | Subgroup |
|---|---|---|
| 111 | != 00 | 32-bit T32 instruction encoding |
| != 111 | - | 16-bit T32 instruction encoding |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | Instruction |
|---|---|---|
| 111 | 00 | B T2 |

## C2.2 32-bit T32 instruction encoding

This section describes the encoding of the 32-bit T32 instruction encoding. This section is decoded from Top level T32 instruction set encoding.

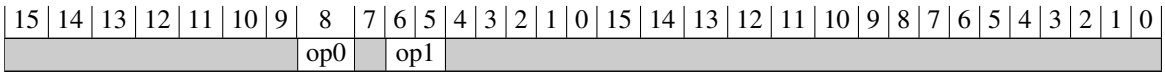



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | op2 | Subgroup |
|---|---|---|---|
| 0100 | - | - | Load/store (multiple, dual, exclusive, acquire-release) |
| 10xx | - | 1 | Branches and miscellaneous control |
| 10x0 | - | 0 | Data-processing (modified immediate) |
| 1101 | 0xxxx | - | Data-processing (register) |
| 1100 | != 1xxx0 | - | Load/store single |
| x11x | - | - | Coprocessor, floating-point, and vector instructions |
| 1101 | 11xxx | - | Long multiply and divide |
| 0101 | - | - | Data-processing (shifted register) |
| 1100 | 1xxx0 | - | UNALLOCATED |
| 10x1 | - | 0 | Data-processing (plain binary immediate) |
| 1101 | 10xxx | - | Multiply, multiply accumulate, and absolute difference |

### C2.2.1 Load/store (multiple, dual, exclusive, acquire-release)

This section describes the encoding of the Load/store (multiple, dual, exclusive, acquire-release). This section is decoded from 32-bit T32 instruction encoding.

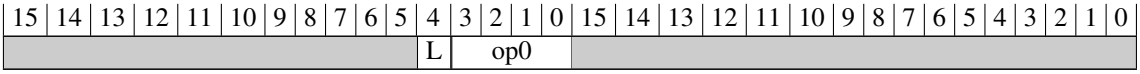



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|
| 0 | 11 | Load/store dual (post-indexed) |
| - | 0x | Load/store multiple |
| 1 | 10 | Load/store dual (literal and immediate) |
| 0 | 10 | Load/store exclusive, load-acquire/store-release |
| 1 | 11 | Load/store dual (pre-indexed), secure gateway |

#### C2.2.1.1 Load/store dual (post-indexed)

This section describes the encoding of the Load/store dual (post-indexed). This section is decoded from Load/store (multiple, dual, exclusive, acquire-release).




**This table shows the decode field values and the associated subgroups:**

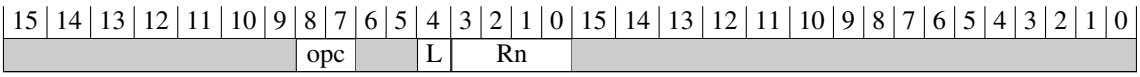
| L | op0 | Subgroup |
|---|---|---|
| - | 1111 | UNPREDICTABLE |

**This table shows the decode field values and the associated instrutions:**

| L | op0 | Instruction |
|---|---|---|
| 1 | != 1111 | LDRD (immediate) T1 |
| 0 | != 1111 | STRD (immediate) T1 |

### C2.2.1.2 Load/store multiple

This section describes the encoding of the Load/store multiple. This section is decoded from Load/store (multiple, dual, exclusive, acquire-release).




**This table shows the decode field values and the associated subgroups:**

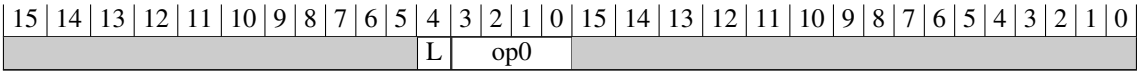
| L | Rn | opc | Subgroup |
|---|---|---|---|
| - | - | 00 | UNALLOCATED |
| - | - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | Rn | opc | Instruction |
|---|---|---|---|
| Alias | | | POP (multiple registers) T2 |
| 1 | 1111 | 01 | CLRM T1 |
| 0 | - | 10 | STMDB, STMFD T1 |
| 1 | != 1111 | 01 | LDM, LDMIA, LDMFD T2 |
| 1 | - | 10 | LDMDB, LDMEA T1 |
| 0 | - | 01 | STM, STMIA, STMEA T2 |
| Alias | | | PUSH (multiple registers) T1 |

### C2.2.1.3 Load/store dual (literal and immediate)

This section describes the encoding of the Load/store dual (literal and immediate). This section is decoded from Load/store (multiple, dual, exclusive, acquire-release).

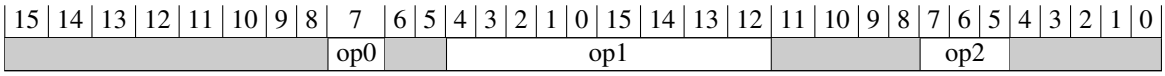



**This table shows the decode field values and the associated instrutions:**

| L | op0 | Instruction |
|---|---|---|
| - | 1111 | LDRD (literal) T1 |
| 1 | != 1111 | LDRD (immediate) T1 |
| 0 | != 1111 | STRD (immediate) T1 |

### C2.2.1.4 Load/store exclusive, load-acquire/store-release

This section describes the encoding of the Load/store exclusive, load-acquire/store-release. This section is decoded from Load/store (multiple, dual, exclusive, acquire-release).




**This table shows the decode field values and the associated subgroups:**

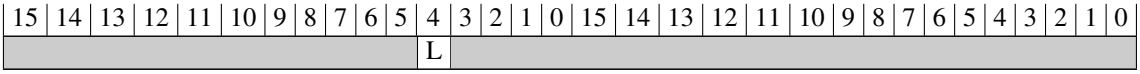
| op0 | op1 | op2 | Subgroup |
|---|---|---|---|
| 1 | - | 1xx | Load-acquire / Store-release |
| 0 | != 0xxxx1111 | - | Load/store exclusive |
| 1 | 0xxxxxxx | 000 | UNALLOCATED |
| 1 | - | 01x | Load/store exclusive byte/half/dual |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | Instruction |
|---|---|---|---|
| 0 | 0xxxx1111 | - | TT, TTT, TTA, TTAT T1 |
| 1 | 1xxxxxxx | 000 | TBB, TBH T1 |

#### *C2.2.1.4.1 Load/store exclusive*

This section describes the encoding of the Load/store exclusive. This section is decoded from Load/store exclusive, load-acquire/store-release.

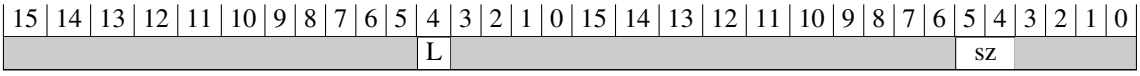



**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| 1 | LDREX T1 |
| 0 | STREX T1 |

#### C2.2.1.4.2 Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual. This section is decoded from Load/store exclusive, load-acquire/store-release.




**This table shows the decode field values and the associated subgroups:**

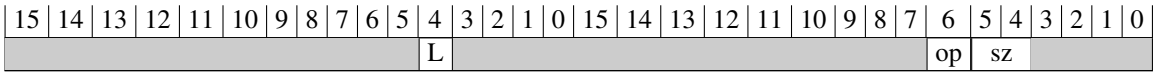
| L | sz | Subgroup |
|---|---|---|
| 1 | 11 | UNALLOCATED |
| 0 | 10 | UNALLOCATED |
| 0 | 11 | UNALLOCATED |
| 1 | 10 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | sz | Instruction |
|---|---|---|
| 0 | 01 | STREXH T1 |
| 0 | 00 | STREXB T1 |
| 1 | 01 | LDREXH T1 |
| 1 | 00 | LDREXB T1 |

#### C2.2.1.4.3 Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release. This section is decoded from Load/store exclusive, load-acquire/store-release.



**This table shows the decode field values and the associated subgroups:**

| L | op | sz | Subgroup |
|---|---|---|---|
| 1 | 1 | 11 | UNALLOCATED |
| 0 | 0 | 11 | UNALLOCATED |
| 0 | 1 | 11 | UNALLOCATED |
| 1 | 0 | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | op | sz | Instruction |
|---|---|---|---|
| 1 | 0 | 10 | LDA T1 |
| 1 | 0 | 00 | LDAB T1 |
| 1 | 1 | 00 | LDAEXB T1 |
| 1 | 1 | 10 | LDAEX T1 |
| 1 | 1 | 01 | LDAEXH T1 |
| 0 | 0 | 10 | STL T1 |
| 1 | 0 | 01 | LDAH T1 |
| 0 | 1 | 00 | STLEXB T1 |
| 0 | 1 | 10 | STLEX T1 |
| 0 | 1 | 01 | STLEXH T1 |
| 0 | 0 | 01 | STLH T1 |
| 0 | 0 | 00 | STLB T1 |

### C2.2.1.5 Load/store dual (pre-indexed), secure gateway

This section describes the encoding of the Load/store dual (pre-indexed), secure gateway. This section is decoded from Load/store (multiple, dual, exclusive, acquire-release).

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | op0 | | | op1 | op2 | | | | op3 | | | | | | | | | | | | | | | | |

**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | op2 | op3 | Subgroup |
|---|---|---|---|---|
| 1 | 1 | 1111 | - | UNPREDICTABLE |
| 0 | 1 | 1111 | != 1110100101111111 | UNPREDICTABLE |
| 1 | 0 | 1111 | - | UNPREDICTABLE |
| 0 | 0 | 1111 | - | UNPREDICTABLE |

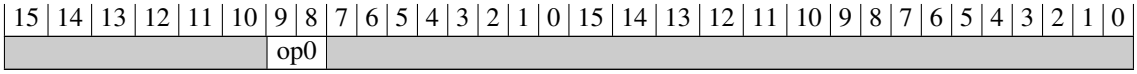
**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | Instruction |
|---|---|---|---|---|
| 0 | 1 | 1111 | 1110100101111111 | SG T1 |
| - | 1 | != 1111 | - | LDRD (immediate) T1 |

| | | | | |
|---|---|---|---|---|
| - | 0 | != 1111 | - | STRD (immediate) T1 |

## C2.2.2 Coprocessor, floating-point, and vector instructions

This section describes the encoding of the Coprocessor, floating-point, and vector instructions. This section is decoded from 32-bit T32 instruction encoding.

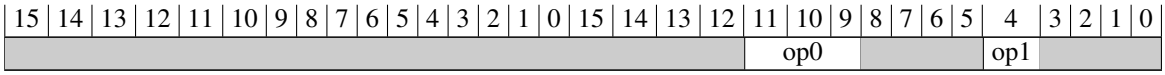



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| 10 | Floating-point and vector miscellaneous instructions |
| != 11 | Floating-point and vector load/store, move, and complex arithmetic instructions |
| 11 | Miscellaneous vector arithmetic instructions |
| 0x | Floating-point and vector load/store, move, and coprocessor instructions |

### C2.2.2.1 Floating-point and vector miscellaneous instructions

This section describes the encoding of the Floating-point and vector miscellaneous instructions. This section is decoded from Coprocessor, floating-point, and vector instructions.

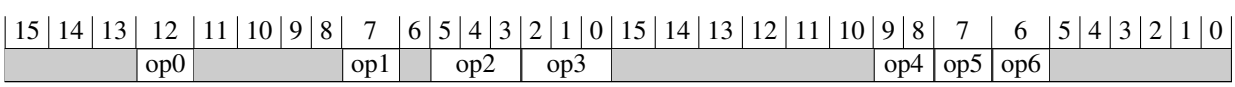



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|
| 101 | 0 | Floating-point data-processing, minNum/maxNum, and convert |
| 101 | 1 | Floating-point and vector move and coprocessor (register) |
| 111 | 0 | Vector immediate and register, and coprocessor data-processing instructions |

#### *C2.2.2.1.1 Floating-point data-processing, minNum/maxNum, and convert*

This section describes the encoding of the Floating-point data-processing, minNum/maxNum, and convert. This section is decoded from Floating-point and vector miscellaneous instructions.

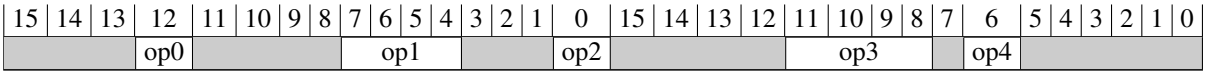


**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | Instruction |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 00x | - | - | - | 1 | VMINNM T2 |
| 1 | 1 | 111 | 110 | - | - | 1 | VCVTP T1 |
| 0 | 1 | 00x | - | - | - | 0 | VDIV T1 |
| 0 | 0 | 11x | - | - | - | 0 | VADD T2 |
| 0 | 0 | 01x | - | - | - | 0 | VNMLS T1 |
| 1 | 1 | 111 | 010 | - | 0 | 1 | VRINTP T1 |
| 0 | 1 | 110 | 000 | 1x | 0 | 1 | VMOV (register) T2 |
| 0 | 1 | 111 | 000 | - | - | 1 | VCVT (integer to floating-point) T1 |
| 0 | 1 | 10x | - | - | - | 1 | VFMS T2 |
| 0 | 1 | 01x | - | - | - | 0 | VFNMS T1 |
| 0 | 1 | 110 | 100 | - | 1 | 1 | VCMPE T1 |
| 0 | 1 | 110 | 101 | - | 1 | 1 | VCMPE T2 |
| 0 | 1 | 110 | 01x | 1x | 1 | 1 | VCVTT T1 |
| 1 | 1 | 111 | 011 | - | 0 | 1 | VRINTM T1 |
| 1 | 1 | 111 | 000 | - | 0 | 1 | VRINTA T1 |
| 0 | 1 | 110 | 001 | - | 0 | 1 | VNEG T2 |
| 0 | 1 | 110 | 111 | - | 0 | 1 | VRINTX T1 |
| 1 | 1 | 111 | 111 | - | - | 1 | VCVTM T1 |
| 0 | 1 | 110 | 01x | 1x | 0 | 1 | VCVTB T1 |
| 1 | 1 | 00x | - | - | - | 0 | VMAXNM T2 |
| 0 | 0 | 00x | - | - | - | 0 | VMLA T2 |
| 0 | 0 | 00x | - | - | - | 1 | VMLS T2 |
| 0 | 1 | 111 | 10x | - | - | 1 | VCVT (floating-point to integer) T1 |
| 0 | 1 | 01x | - | - | - | 1 | VFNMA T1 |
| 0 | 1 | 10x | - | - | - | 0 | VFMA T2 |
| 0 | 1 | 110 | 000 | - | 1 | 1 | VABS T2 |
| 0 | 1 | 110 | 110 | - | 0 | 1 | VRINTR T1 |
| 0 | 1 | 11x | - | - | - | 0 | VMOV (immediate) T2 |
| 0 | 1 | 110 | 100 | - | 0 | 1 | VCMP T1 |
| 1 | 1 | 110 | 000 | 10 | 0 | 1 | VMOVX T1 |
| 0 | 1 | 110 | 101 | - | 0 | 1 | VCMP T2 |
| 0 | 1 | 110 | 111 | 1x | 1 | 1 | VCVT (between double-precision and single-precision) T1 |
| 0 | 1 | 111 | x1x | - | - | 1 | VCVT (between floating-point and fixed-point) T1 |
| 1 | 1 | 110 | 000 | 10 | 1 | 1 | VINS T1 |
| 1 | 1 | 111 | 001 | - | 0 | 1 | VRINTN T1 |
| 1 | 0 | - | - | - | - | 0 | VSEL T1 |
| 0 | 0 | 01x | - | - | - | 1 | VNMLA T1 |
| 0 | 1 | 110 | 110 | - | 1 | 1 | VRINTZ T1 |
| 1 | 1 | 111 | 101 | - | - | 1 | VCVTN T1 |
| 0 | 0 | 10x | - | - | - | 0 | VMUL T2 |
| 1 | 1 | 111 | 100 | - | - | 1 | VCVTA T1 |
| 0 | 1 | 110 | 001 | - | 1 | 1 | VSQRT T1 |
| 0 | 1 | 111 | 10x | - | 0 | 1 | VCVTR T1 |
| 0 | 0 | 01x | - | - | - | - | VNMUL T2 |
| 0 | 0 | 11x | - | - | - | 1 | VSUB T2 |

#### C2.2.2.1.2 Floating-point and vector move and coprocessor (register)

This section describes the encoding of the Floating-point and vector move and coprocessor (register). This section is decoded from Floating-point and vector miscellaneous instructions.

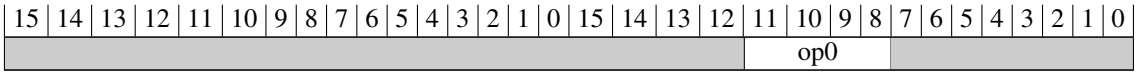



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | Instruction |
|---|---|---|---|---|---|
| 0 | 1110 | - | 1010 | - | VMSR T1 |
| 0 | xxx1 | - | 1011 | - | VMOV (vector lane to general-purpose register) T1 |
| 0 | 1111 | - | 1010 | - | VMRS T1 |
| 1 | xxx1 | - | != 1xxx | - | MRC, MRC2 T2 |
| 0 | xxx1 | - | != 1xxx | - | MRC, MRC2 T1 |
| 0 | 0xx0 | - | 1011 | - | VMOV (general-purpose register to vector lane) T1 |
| 0 | 000x | - | 1010 | - | VMOV (between general-purpose register and single-precision register) T1 |
| 0 | xxx0 | - | != 1xxx | - | MCR, MCR2 T1 |
| 1 | xxx0 | - | != 1xxx | - | MCR, MCR2 T2 |
| 0 | 1x10 | 0 | 1011 | 0 | VDUP T1 |

#### C2.2.2.1.3 Vector immediate and register, and coprocessor data-processing instructions

This section describes the encoding of the Vector immediate and register, and coprocessor data-processing instructions. This section is decoded from Floating-point and vector miscellaneous instructions.

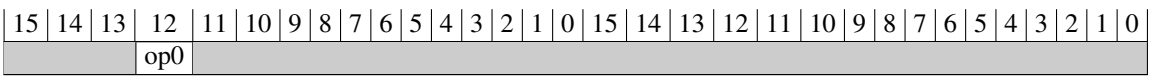



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| != 1111 | Coprocessor data-processing instructions |
| 1110 | Miscellaneous vector register instructions |
| 1111 | Miscellaneous vector register and immediate instructions |

#### Coprocessor data-processing instructions

This section describes the encoding of the Coprocessor data-processing instructions. This section is decoded from Vector immediate and register, and coprocessor data-processing instructions.



**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 1 | CDP, CDP2 T2 |
| 0 | CDP, CDP2 T1 |

### *Miscellaneous vector register instructions*

This section describes the encoding of the Miscellaneous vector register instructions. This section is decoded from Vector immediate and register, and coprocessor data-processing 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | op0 | | | | | op1 | op2 | | | op3 | | op4 | | | | | op5 | | | | | | op6 | | | | | | op7 |

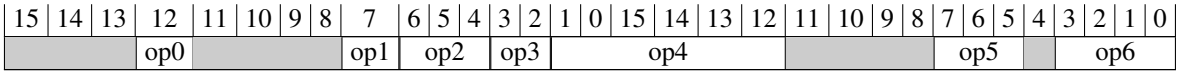
**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | op7 | Instruction |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | x11 | != 11 | 01 | - | 10x | 1 | VQMOVUN T1 |
| - | 0 | != x11 | - | x1 | 1 | x10 | - | VMLAS (vector by vector plus scalar) T1 |
| - | 0 | x11 | != 11 | 11 | 1 | 10x | 1 | VMIN, VMINA T2 |
| 1 | 0 | != x11 | - | x0 | - | x0x | 0 | VQDMLSDH, VQRDMLSDH T1 |
| Alias | | | | | | | | VMLALV T1 |
| - | 0 | x11 | != 11 | 01 | 1 | 111 | - | VQSHL, VQSHLU T1 |
| - | 0 | x11 | 11 | 11 | 1 | 10x | 1 | VMINNM, VMINNMA (floating-point) T2 |
| - | 0 | x11 | != 11 | 11 | 0 | 10x | 1 | VMAX, VMAXA T2 |
| - | 1 | 111 | - | - | - | x0x | 0 | VMLADAV T1 |
| - | 0 | x11 | - | x1 | - | x0x | 0 | VMULL (polynomial) T1 |
| 1 | 0 | x11 | != 11 | 01 | - | 10x | 1 | VMOVN T1 |
| - | 0 | != x11 | - | x1 | 0 | x0x | 1 | VMULH, VRMULH T1 |
| - | 0 | != x11 | - | x1 | 1 | x0x | 1 | VMULH, VRMULH T2 |
| 0 | 0 | != x11 | - | x1 | 1 | x11 | - | VMUL (vector) T2 |
| - | 0 | != x11 | - | x0 | 0 | x11 | - | VQDMLAH, VQRDMLAH (vector by scalar plus vector) T1 |
| - | 0 | != x11 | - | x0 | 0 | x10 | - | VQDMLAH, VQRDMLAH (vector by scalar plus vector) T2 |
| - | 0 | x11 | - | x1 | 0 | x10 | - | VFMA (vector by scalar plus vector, floating-point) T1 |
| 0 | 0 | != x11 | - | x0 | - | x0x | 1 | VQDMLADH, VQRDMLADH T2 |
| - | 1 | != 111 | - | - | - | x0x | 0 | VMLALDAV T1 |
| - | 0 | x11 | - | x1 | 0 | x11 | - | VMUL (floating-point) T2 |
| 0 | 1 | != 111 | - | - | - | x0x | 1 | VMLSLDAV T1 |
| - | 0 | x11 | - | x1 | 1 | x10 | - | VFMAS (vector by vector plus scalar, floating-point) T1 |
| 0 | 1 | 111 | - | - | - | x0x | 1 | VMLSDAV T1 |
| 1 | 1 | 111 | - | x0 | - | x0x | 1 | VMLSDAV T2 |
| 0 | 0 | != x11 | - | x1 | 0 | x11 | - | VQDMULH, VQRDMULH T3 |
| - | 0 | x11 | - | x0 | - | x0x | - | VCMUL (floating-point) T1 |
| 0 | 0 | != x11 | - | x0 | - | x0x | 0 | VQDMLADH, VQRDMLADH T1 |
| 1 | 0 | != x11 | - | x1 | 0 | x11 | - | VQDMULH, VQRDMULH T4 |
| 1 | 0 | != x11 | - | x0 | - | x0x | 1 | VQDMLSDH, VQRDMLSDH T2 |
| - | 0 | != x11 | - | x0 | 1 | x11 | - | VQDMLASH, VQRDMLASH (vector by vector plus scalar) T1 |
| - | 0 | != x11 | - | x0 | 1 | x10 | - | VQDMLASH, VQRDMLASH (vector by vector plus scalar) T2 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | 0 | x11 | != 11 | 01 | 1 | 011 | - | VSHL T2 |
| - | 0 | x11 | != 11 | 11 | 1 | 011 | - | VRSHL T2 |
| - | 0 | x11 | != 11 | 01 | - | 00x | 1 | VSHLL T2 |
| - | 0 | x11 | 11 | 11 | - | 00x | 1 | VCVT (between single and half-precision floating-point) T1 |
| Alias | | | | | | | | VMLAV T1 |
| 1 | 1 | != 111 | - | x0 | - | x0x | 1 | VRMLSLDAVH T1 |
| - | 0 | x11 | != 11 | 11 | - | 00x | 1 | VQMOVN T1 |
| - | 0 | != x11 | - | x1 | - | x0x | 0 | VMULL (integer) T1 |
| 1 | 0 | != x11 | - | x1 | 1 | x11 | - | VBRSR T1 |
| - | 0 | x11 | 11 | 11 | 0 | 10x | 1 | VMAXNM, VMAXNMA (floating-point) T2 |
| - | 0 | x11 | != 11 | 11 | 1 | 111 | - | VQRSHL T2 |
| - | 0 | != x11 | - | x1 | 0 | x10 | - | VMLA (vector by scalar plus vector) T1 |

### *Miscellaneous vector register and immediate instructions*

This section describes the encoding of the Miscellaneous vector register and immediate instructions. This section is decoded from Vector immediate and register, and coprocessor data-processing instructions.



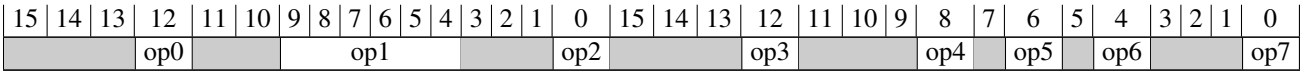

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | Instruction |
|---|---|---|---|---|---|---|---|
| - | 1 | 110 | != 11 | 00xxxx | 10x | xxx0 | VMINV, VMINAV T2 |
| 0 | 0 | != x11 | - | x1xxx0 | x11 | 111x | VIDUP, VIWDUP T2 |
| 0 | 0 | != x11 | - | x1xxx0 | x11 | - | VIDUP, VIWDUP T1 |
| 0 | 1 | x0x | - | - | 11x | xxx1 | VSHRN T1 |
| - | 0 | x11 | - | x0xxxx | x0x | xxx1 | VQDMULL T1 |
| - | 0 | != x11 | - | x0xxx0 | x11 | - | VQADD T2 |
| - | 0 | x11 | - | x0xxx0 | x10 | - | VADD (floating-point) T2 |
| 1 | 0 | != x11 | - | x1xxx0 | x10 | - | VPT T4 |
| 1 | 0 | != x11 | - | x1xxx1 | x1x | - | VPT T6 |
| 1 | 0 | != x11 | - | x1xxx0 | x0x | xxx0 | VPT T1 |
| 1 | 0 | != x11 | - | x1xxx1 | x0x | - | VPT T3 |
| 1 | 0 | != x11 | - | x1xxx0 | x0x | xxx1 | VPT T2 |
| 0 | 0 | != x11 | - | x1xxx1 | x10 | - | VSUB (vector) T2 |
| 0 | 0 | != x11 | - | x1xxx0 | x10 | - | VADD (vector) T2 |
| - | 1 | 110 | != 11 | 10xxxx | 10x | xxx0 | VMINV, VMINAV T1 |
| 0 | 0 | != x11 | - | x0xxxx | x0x | xxx0 | VHCADD T1 |
| - | 1 | x1x | x0 | 00xxxx | 01x | xxx0 | VMOVL T1 |
| - | 0 | x11 | - | x1xxxx | x0x | - | VPT (floating-point) T1 |
| 1 | 1 | x0x | - | - | 11x | xxx0 | VQRSHRUN T1 |
| - | 1 | 110 | 11 | 10xxxx | 10x | xxx0 | VMINNMV, VMINNMAV (floating-point) T1 |
| 0 | 0 | != x11 | - | x1xxx1 | x11 | 111x | VDDUP, VDWDUP T2 |
| - | 1 | 111 | - | x0xxxx | x0x | xxx0 | VMLADAV T2 |
| 0 | 0 | != x11 | - | x1xxx1 | x11 | - | VDDUP, VDWDUP T1 |
| 1 | 0 | != x11 | - | x0xxxx | x0x | xxx0 | VCADD T1 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| - | 0 | != x11 | - | x0xxx0 | x10 | - | VHADD T2 |
| - | 0 | x11 | - | x0xxx1 | x10 | - | VSUB (floating-point) T2 |
| - | 1 | != 111 | - | 01xxxx | 00x | xxx0 | VADDLV T1 |
| 1 | 0 | != 011 | - | x10001 | x0x | - | VCMP (vector) T3 |
| - | 0 | != x11 | - | x0xxx1 | x10 | - | VHSUB T2 |
| 1 | 0 | 011 | - | x1000x | x1x | 1101 | VPNOT T1 |
| - | 1 | 111 | != 11 | 01xxxx | 00x | xxx0 | VADDV T1 |
| 1 | 0 | x11 | - | x1xxx0 | x0x | xxx1 | VPSEL T1 |
| - | 1 | 110 | 11 | 00xxxx | 10x | xxx0 | VMINNMV, VMINNMAV (floating-point) T2 |
| 1 | 0 | != 011 | - | x10000 | x0x | xxx1 | VCMP (vector) T2 |
| - | 0 | 011 | - | x1000x | x1x | != 1101 | VCMP (floating-point) T2 |
| - | 0 | 011 | - | x1000x | x0x | - | VCMP (floating-point) T1 |
| 1 | 0 | != 011 | - | x10000 | x0x | xxx0 | VCMP (vector) T1 |
| 1 | 0 | != 011 | - | x10001 | x1x | - | VCMP (vector) T6 |
| 1 | 0 | != 011 | - | x10000 | x10 | - | VCMP (vector) T4 |
| 1 | 0 | != 011 | - | x10000 | x11 | - | VCMP (vector) T5 |
| 1 | 0 | x11 | - | x1xxxx | x1x | 1101 | VPST T1 |
| Alias | | | | | | | VRMLALVH T1 |
| - | 1 | x1x | - | - | 01x | xxx0 | VSHLL T1 |
| 0 | 1 | x1x | - | xxxxx0 | 110 | - | VSHLC T1 |
| 0 | 1 | x0x | - | - | 11x | xxx0 | VQSHRUN T1 |
| 1 | 0 | != x11 | - | x1xxx0 | x11 | - | VPT T5 |
| - | 1 | x0x | - | - | 01x | xxx0 | VQSHRN T1 |
| 1 | 1 | x0x | - | - | 11x | xxx1 | VRSHRN T1 |
| 0 | 0 | x11 | - | x0xxxx | x0x | xxx0 | VADC T1 |
| - | 1 | 110 | 11 | 10xxxx | 00x | xxx0 | VMAXNMV, VMAXNMAV (floating-point) T1 |
| - | 1 | 110 | != 11 | 00xxxx | 00x | xxx0 | VMAXV, VMAXAV T2 |
| - | 1 | 110 | != 11 | 10xxxx | 00x | xxx0 | VMAXV, VMAXAV T1 |
| - | 1 | 110 | 11 | 00xxxx | 00x | xxx0 | VMAXNMV, VMAXNMAV (floating-point) T2 |
| - | 1 | != 11x | - | x0xxxx | x0x | xxx0 | VRMLALDAVH T1 |
| - | 0 | != x11 | - | x0xxx1 | x11 | - | VQSUB T2 |
| - | 0 | x11 | - | x0xxxx | x11 | - | VQDMULL T2 |
| 1 | 0 | x11 | - | x0xxxx | x0x | xxx0 | VSBC T1 |
| - | 1 | x0x | - | - | 01x | xxx1 | VQRSHRN T1 |
| - | 1 | != 011 | - | x0xxxx | x0x | xxx1 | VABAV T1 |
| - | 0 | x11 | - | x1xxxx | x1x | != 1101 | VPT (floating-point) T2 |

### C2.2.2.2 Floating-point and vector load/store, move, and complex arithmetic instructions

This section describes the encoding of the Floating-point and vector load/store, move, and complex arithmetic instructions. This section is decoded from Coprocessor, floating-point, and vector instructions.

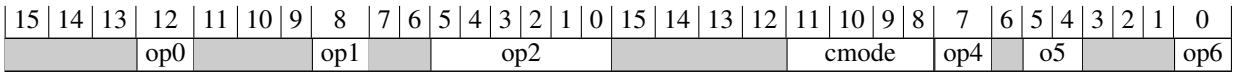



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | op7 | Instruction |
|---|---|---|---|---|---|---|---|---|
| 0 | 01xx01 | - | - | 1 | - | - | - | VLDR T3 |
| 1 | 0xxx1x | 0 | 0 | 0 | 1 | 0 | 0 | VCMLA (floating-point) T1 |
| 0 | 10000x | - | - | 1 | - | 1 | - | VMOV (between general-purpose register and half-precision register) T1 |
| 0 | 01xx00 | - | - | 1 | - | - | - | VSTR T3 |
| 1 | 0x1x0x | 0 | 0 | 0 | 1 | 0 | 0 | VCADD (floating-point) T1 |

### C2.2.2.3 Miscellaneous vector arithmetic instructions

This section describes the encoding of the Miscellaneous vector arithmetic instructions. This section is decoded from Coprocessor, floating-point, and vector instructions.



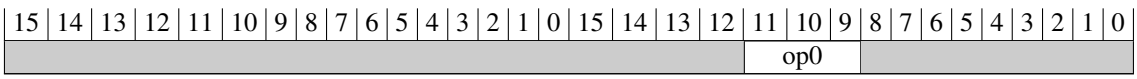

**This table shows the decode field values and the associated instrutions:**

| o5:cmode | op0 | op1 | op2 | op3 | op4 | op5 | op6 | Instruction |
|---|---|---|---|---|---|---|---|---|
| - | 1 | 1 | 11xx00 | 0100 | 0 | x0 | 0 | VCLS T1 |
| - | 0 | 0 | 00xxx0 | 0001 | - | x1 | 0 | VAND T1 |
| x1111x<br>x0xxx0<br>x011x1 | - | 1 | 000xxx | - | 0 | x1 | - | VMOV (immediate) (vector) T1 |
| - | - | 0 | xxxxx0 | 0000 | - | x1 | 0 | VQADD T1 |
| Alias | | | | | | | | VORN (immediate) T1 |
| - | - | 0 | != 11xxx0 | 0100 | - | x0 | 0 | VSHL T3 |
| - | - | 1 | 000xxx | 0xx1<br>10x1<br>1101 | 0 | 11 | - | VBIC (immediate) T1 |
| - | - | 0 | != 11xxx0 | 0110 | - | x1 | 0 | VMIN, VMINA T1 |
| - | 0 | 0 | 0xxxx0 | 1101 | - | x0 | 0 | VADD (floating-point) T1 |
| - | 1 | 0 | 1xxxx0 | 1111 | - | x1 | 0 | VMINNM, VMINNMA (floating-point) T1 |
| - | - | 1 | != 000xxx | 0111 | 0 | x1 | 0 | VQSHL, VQSHLU T2 |
| - | - | 0 | != 11xxx0 | 0110 | - | x0 | 0 | VMAX, VMAXA T1 |
| - | 0 | 0 | xxxxx0 | 1000 | - | x0 | 0 | VADD (vector) T1 |
| - | - | 1 | != 000xxx | 0010 | 0 | x1 | 0 | VRSHR T1 |
| - | - | 0 | xxxxx0 | 0111 | - | x0 | 0 | VABD T1 |
| - | - | 1 | != 000xxx | 0000 | 0 | x1 | 0 | VSHR T1 |
| - | - | 1 | 000xxx | 0xx1<br>10x1 | 0 | 01 | - | VORR (immediate) T1 |
| Alias | | | | | | | | VMOV (register) (vector) T1 |
| - | 1 | 0 | xxxxx0 | 1000 | - | x0 | 0 | VSUB (vector) T1 |
| - | 0 | 0 | 1xxxx0 | 1100 | - | x1 | 0 | VFMA, VFMS (floating-point) T2 |
| - | 0 | 0 | 0xxxx0 | 1100 | - | x1 | 0 | VFMA, VFMS (floating-point) T1 |
| - | - | 1 | != 000xxx | 11xx | 0 | x1 | 0 | VCVT (between floating-point and fixed-point) (vector) T1 |
| - | 1 | 1 | 11xx00 | 0000 | 0 | x0 | 0 | VREV64 T1 |
| - | 0 | 0 | xxxxx0 | 1001 | - | x1 | 0 | VMUL (vector) T1 |
| - | 0 | 0 | 1xxxx0 | 1101 | - | x0 | 0 | VSUB (floating-point) T1 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | - | 0 | xxxxx0 | 0000 | - | x0 | 0 | VHADD T1 |
| - | 1 | 0 | 0xxxx0 | 1101 | - | x1 | 0 | VMUL (floating-point) T1 |
| - | 1 | 1 | 11xx01 | 0111 | 0 | x0 | 0 | VABS (floating-point) T1 |
| - | 1 | 1 | 11xx00 | 0111 | 1 | x0 | 0 | VQNEG T1 |
| - | - | 0 | xxxxx0 | 0010 | - | x0 | 0 | VHSUB T1 |
| - | 1 | 1 | 110000 | 0101 | 1 | x0 | 0 | VMVN (register) T1 |
| - | - | 0 | != 11xxx0 | 0101 | - | x0 | 0 | VRSHL T1 |
| - | 1 | 0 | xxxxx0 | 1011 | - | x0 | 0 | VQDMULH, VQRDMULH T2 |
| Alias | | | | | | | | VAND (immediate) T1 |
| - | 1 | 1 | 11xx00 | 0111 | 0 | x0 | 0 | VQABS T1 |
| - | 1 | 1 | 11xx01 | 0011 | 0 | x0 | 0 | VABS (vector) T1 |
| - | 1 | 1 | 11xx10 | 01xx | - | x0 | 0 | VRINT (floating-point) T1 |
| - | 1 | 0 | 1xxxx0 | 1101 | - | x0 | 0 | VABD (floating-point) T1 |
| - | 1 | 1 | 11xx11 | 011x | - | x0 | 0 | VCVT (between floating-point and integer) T1 |
| - | 0 | 1 | != 000xxx | 0101 | 0 | x1 | 0 | VSHL T1 |
| - | - | 0 | xxxxx0 | 0001 | - | x0 | 0 | VRHADD T1 |
| - | 0 | 0 | 11xxx0 | 0001 | - | x1 | 0 | VORN T1 |
| - | 1 | 1 | != 000xxx | 0101 | 0 | x1 | 0 | VSLI T1 |
| - | 1 | 1 | != 000xxx | 0100 | 0 | x1 | 0 | VSRI T1 |
| - | 0 | 0 | 10xxx0 | 0001 | - | x1 | 0 | VORR T1 |
| - | 1 | 0 | 00xxx0 | 0001 | - | x1 | 0 | VEOR T1 |
| - | 0 | 0 | xxxxx0 | 1011 | - | x0 | 0 | VQDMULH, VQRDMULH T1 |
| - | 1 | 1 | 11xx11 | 00xx | - | x0 | 0 | VCVT (from floating-point to integer) T1 |
| - | 1 | 1 | 11xx01 | 0011 | 1 | x0 | 0 | VNEG (vector) T1 |
| - | - | 1 | 000xxx | xx00<br>x010<br>0110 | 0 | 11 | - | VMVN (immediate) T1 |
| - | - | 0 | != 11xxx0 | 0100 | - | x1 | 0 | VQSHL, VQSHLU T4 |
| - | 1 | 1 | 11xx01 | 0111 | 1 | x0 | 0 | VNEG (floating-point) T1 |
| - | 1 | 1 | 11xx00 | 0001 | 0 | x0 | 0 | VREV16 T1 |
| - | 1 | 0 | 0xxxx0 | 1111 | - | x1 | 0 | VMAXNM, VMAXNMA (floating-point) T1 |
| - | 0 | 0 | 01xxx0 | 0001 | - | x1 | 0 | VBIC (register) T1 |
| - | 1 | 1 | != 000xxx | 0110 | 0 | x1 | 0 | VQSHL, VQSHLU T3 |
| - | 1 | 1 | 11xx00 | 0000 | 1 | x0 | 0 | VREV32 T1 |
| - | - | 0 | xxxxx0 | 0010 | - | x1 | 0 | VQSUB T1 |
| - | 1 | 1 | 11xx00 | 0100 | 1 | x0 | 0 | VCLZ T1 |
| - | - | 0 | != 11xxx0 | 0101 | - | x1 | 0 | VQRSHL T1 |

### C2.2.2.4 Floating-point and vector load/store, move, and coprocessor instructions

This section describes the encoding of the Floating-point and vector load/store, move, and coprocessor instructions. This section is decoded from Coprocessor, floating-point, and vector instructions.

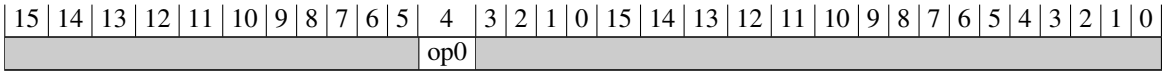



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| 111 | Vector load/store and move instructions |
| 101 | Coprocessor and Floating-point load/store, move, and security |

#### C2.2.2.4.1 Vector load/store and move instructions

This section describes the encoding of the Vector load/store and move instructions. This section is decoded from Floating-point and vector load/store, move, and coprocessor instructions.

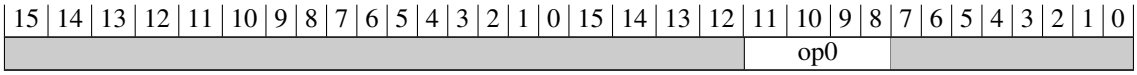



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| 0 | Coprocessor and vector store instructions |
| 1 | Coprocessor and vector load instructions |

##### Coprocessor and vector store instructions

This section describes the encoding of the Coprocessor and vector store instructions. This section is decoded from Vector load/store and move instructions.

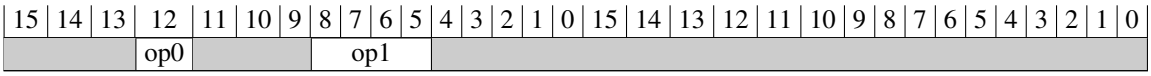



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| != 1xxx | Coprocessor store instructions |
| 111x | Vector store instructions |

Coprocessor store instructions

This section describes the encoding of the Coprocessor store instructions. This section is decoded from Coprocessor and vector store instructions.

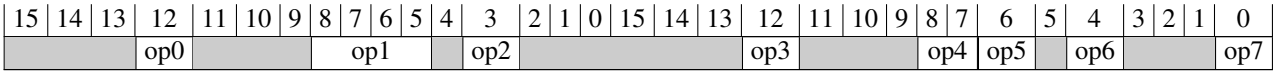



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | Instruction |
|---|---|---|
| 1 | 0010 | MCRR, MCRR2 T2 |
| 0 | 0010 | MCRR, MCRR2 T1 |
| 1 | - | STC, STC2 T2 |
| 0 | - | STC, STC2 T1 |

Vector store instructions

This section describes the encoding of the Vector store instructions. This section is decoded from Coprocessor and vector store instructions.

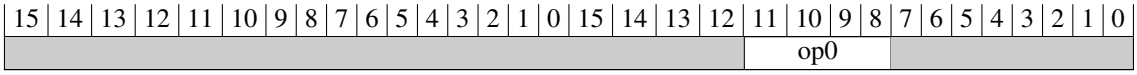



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | op7 | Instruction |
|---|---|---|---|---|---|---|---|---|
| - | - | - | 0 | 11 | - | - | - | VSTR (System Register) T1 |
| - | 01x0 | - | 0 | - | 0 | 1 | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T2 |
| 0 | - | - | 1 | 10 | - | - | - | VSTRB, VSTRH, VSTRW T7 |
| 0 | - | - | 1 | 01 | - | - | - | VSTRB, VSTRH, VSTRW T6 |
| 0 | - | - | 1 | 00 | - | - | - | VSTRB, VSTRH, VSTRW T5 |
| - | xx0x | 1 | 0 | != 11 | - | - | - | VSTRB, VSTRH, VSTRW T2 |
| - | xx0x | 0 | 0 | != 11 | - | - | - | VSTRB, VSTRH, VSTRW T1 |
| - | 01x0 | - | 0 | - | 1 | 1 | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T4 |
| 1 | 1xxx | - | 1 | 0x | - | - | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T5 |
| 1 | 1xxx | - | 1 | 1x | - | - | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T6 |
| 1 | 01xx | - | 1 | != 11 | - | - | 0 | VST2 T1 |
| 0 | 00x0 | - | 0 | 1x | - | - | - | VMOV (two 32 bit vector lanes to two general-purpose registers) T1 |
| - | 01x0 | - | 0 | - | 1 | 0 | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T3 |
| 1 | 01xx | - | 1 | != 11 | - | - | 1 | VST4 T1 |
| - | 01x0 | - | 0 | - | 0 | 0 | - | VSTRB, VSTRH, VSTRW, VSTRD (vector) T1 |

### *Coprocessor and vector load instructions*

This section describes the encoding of the Coprocessor and vector load instructions. This section is decoded from Vector load/store and move instructions.



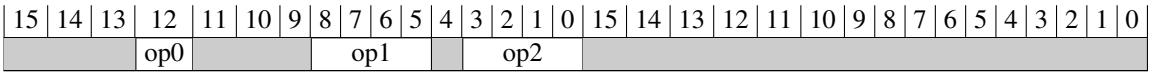

**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| != 1xxx | Coprocessor load instructions |

| | |
|---|---|
| 111x | Vector load instructions |

Coprocessor load instructions

This section describes the encoding of the Coprocessor load instructions. This section is decoded from Coprocessor and vector load instructions.

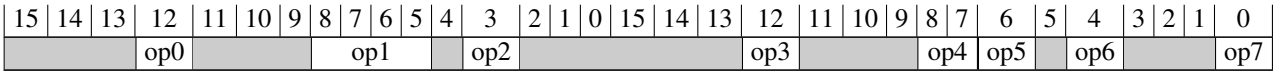



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | Instruction |
|---|---|---|---|
| 1 | - | 1111 | LDC, LDC2 (literal) T2 |
| 1 | 0010 | - | MRRC, MRRC2 T2 |
| 0 | 0010 | - | MRRC, MRRC2 T1 |
| 0 | - | 1111 | LDC, LDC2 (literal) T1 |
| 0 | - | != 1111 | LDC, LDC2 (immediate) T1 |
| 1 | - | != 1111 | LDC, LDC2 (immediate) T2 |

Vector load instructions

This section describes the encoding of the Vector load instructions. This section is decoded from Coprocessor and vector load instructions.

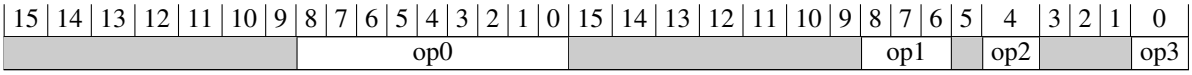



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | op7 | Instruction |
|---|---|---|---|---|---|---|---|---|
| - | xx0x | 1 | 0 | != 11 | - | - | - | VLDRB, VLDRH, VLDRW T2 |
| 1 | 01xx | - | 1 | != 11 | - | - | 1 | VLD4 T1 |
| 1 | 01xx | - | 1 | != 11 | - | - | 0 | VLD2 T1 |
| 0 | - | - | 1 | 01 | - | - | - | VLDRB, VLDRH, VLDRW T6 |
| 0 | - | - | 1 | 10 | - | - | - | VLDRB, VLDRH, VLDRW T7 |
| 0 | - | - | 1 | 00 | - | - | - | VLDRB, VLDRH, VLDRW T5 |
| 1 | 1xxx | - | 1 | 0x | - | - | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T5 |
| - | 01x0 | - | 0 | - | 1 | 1 | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T4 |
| 1 | 1xxx | - | 1 | 1x | - | - | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T6 |
| - | 01x0 | - | 0 | - | 0 | 0 | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T1 |
| - | 01x0 | - | 0 | - | 1 | 0 | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T3 |
| - | 01x0 | - | 0 | - | 0 | 1 | - | VLDRB, VLDRH, VLDRW, VLDRD (vector) T2 |
| - | - | - | 0 | 11 | - | - | - | VLDR (System Register) T1 |
| - | xx0x | 0 | 0 | != 11 | - | - | - | VLDRB, VLDRH, VLDRW T1 |
| 0 | 00x0 | - | 0 | 1x | - | - | - | VMOV (two general-purpose registers to two 32 bit vector lanes) T1 |

#### C2.2.2.4.2 Coprocessor and Floating-point load/store, move, and security

This section describes the encoding of the Coprocessor and Floating-point load/store, move, and security. This section is decoded from Floating-point and vector load/store, move, and coprocessor instructions.

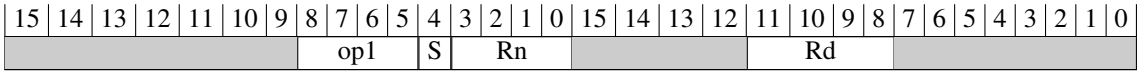



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | Instruction |
|---|---|---|---|---|
| 00x11xxxx | 00x | - | - | VLLDM T1 |
| xxxx1xxxx | 1xx | - | 0 | VLDM T1 |
| 00x11xxxx | 01x | - | - | VLLDM T2 |
| 01x011111 | 1xx | - | 0 | VSCCLRM T1 |
| 01x011111 | 0xx | - | - | VSCCLRM T2 |
| xxxx0xxxx | 1xx | - | 1 | FSTMDBX, FSTMIAX T1 |
| xxxx1xxxx | 1xx | - | 1 | FLDMDBX, FLDMIAX T1 |
| 1xx01xxxx | 0xx | - | - | VLDR T2 |
| 1xx01xxxx | 1xx | - | - | VLDR T1 |
| 0010xxxxx | 000 | 1 | - | VMOV (between two general-purpose registers and two single-precision registers) T1 |
| Alias | | | | VPOP T2 |
| Alias | | | | VPOP T1 |
| xxxx0xxxx | 0xx | - | - | VSTM T2 |
| xxxx0xxxx | 1xx | - | 0 | VSTM T1 |
| xxxx1xxxx | 0xx | - | - | VLDM T2 |
| 0010xxxxx | 100 | 1 | - | VMOV (between two general-purpose registers and a doubleword register) T1 |
| Alias | | | | VPUSH T2 |
| Alias | | | | VPUSH T1 |
| 1xx00xxxx | 0xx | - | - | VSTR T2 |
| 00x10xxxx | 00x | - | - | VLSTM T1 |
| 00x10xxxx | 01x | - | - | VLSTM T2 |
| 1xx00xxxx | 1xx | - | - | VSTR T1 |

## C2.2.3 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate). This section is decoded from 32-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

| Rd | Rn | S | op1 | Subgroup |
|---|---|---|---|---|

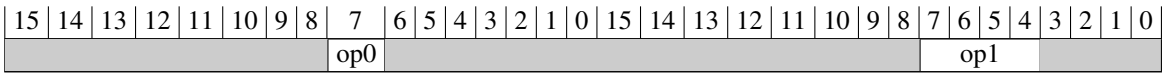
| | | | | |
|---|---|---|---|---|
| - | - | - | 1100 | UNALLOCATED |
| - | - | - | 1111 | UNALLOCATED |
| - | - | - | 0101 | UNALLOCATED |
| - | - | - | 011x | UNALLOCATED |
| - | - | - | 1001 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| Rd | Rn | S | op1 | Instruction |
|---|---|---|---|---|
| != 1111 | - | 1 | 0000 | ANDS (immediate) |
| 1111 | - | 1 | 0100 | TEQ (immediate) T1 |
| != 1111 | != 1101 | 1 | 1101 | SUBS (SP minus immediate) |
| - | != 1111 | - | 0010 | ORR (immediate) T1 |
| - | 1101 | 0 | 1000 | ADD (SP plus immediate) T3 |
| - | != 1101 | 0 | 1000 | ADD (immediate) T3 |
| - | != 1101 | 0 | 1101 | SUB (immediate) T3 |
| 1111 | - | 1 | 1000 | CMN (immediate) T1 |
| - | - | - | 0001 | BIC (immediate) T1 |
| - | - | 0 | 0000 | AND (immediate) T1 |
| - | 1101 | 0 | 1101 | SUB (SP minus immediate) T2 |
| != 1111 | 1101 | 1 | 1101 | SUBS (SP minus immediate) |
| - | != 1111 | - | 0011 | ORN (immediate) T1 |
| 1111 | - | 1 | 1101 | CMP (immediate) T2 |
| - | - | - | 1011 | SBC (immediate) T1 |
| - | 1111 | - | 0011 | MVN (immediate) T1 |
| 1111 | - | 1 | 0000 | TST (immediate) T1 |
| != 1111 | 1101 | 1 | 1000 | ADDS (SP plus immediate) |
| != 1111 | - | 1 | 0100 | EORS (immediate) |
| - | 1111 | - | 0010 | MOV (immediate) T2 |
| - | - | - | 1110 | RSB (immediate) T2 |
| - | - | - | 1010 | ADC (immediate) T1 |
| != 1111 | != 1101 | 1 | 1000 | ADDS (immediate) |
| - | - | 0 | 0100 | EOR (immediate) T1 |

## C2.2.4 Data-processing (register)

This section describes the encoding of the Data-processing (register). This section is decoded from 32-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|

| | | |
|---|---|---|
| 0 | 1xxx | Register extends |
| 1 | 0xxx | Parallel add-subtract |
| 1 | 10xx | Data-processing (two source registers) |
| 1 | 11xx | UNALLOCATED |
| 0 | 0001 | UNALLOCATED |
| 0 | 001x | UNALLOCATED |
| 0 | 01xx | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | Instruction |
|---|---|---|
| 0 | 0000 | MOV, MOVS (register-shifted register) T2 |
| Alias | | ROR (register) T2 |
| Alias | | LSR (register) T2 |
| Alias | | ASR (register) T2 |
| Alias | | ASRS (register) T2 |
| Alias | | LSRS (register) T2 |
| Alias | | LSLS (register) T2 |
| Alias | | LSL (register) T2 |
| Alias | | RORS (register) T2 |

### C2.2.4.1 Register extends

This section describes the encoding of the Register extends. This section is decoded from 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | op1 | | U | Rn | | | | | | | | | | | | | | | | | | | |

**This table shows the decode field values and the associated subgroups:**

| Rn | U | op1 | Subgroup |
|---|---|---|---|
| - | - | 11 | UNALLOCATED |

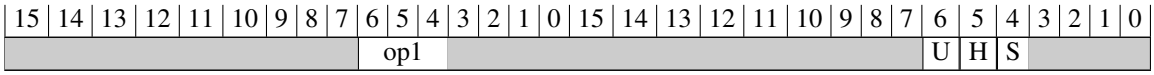
**This table shows the decode field values and the associated instrutions:**

| Rn | U | op1 | Instruction |
|---|---|---|---|
| 1111 | 0 | 01 | SXTB16 T1 |
| 1111 | 1 | 01 | UXTB16 T1 |
| != 1111 | 1 | 00 | UXTAH T1 |
| != 1111 | 0 | 00 | SXTAH T1 |
| != 1111 | 0 | 10 | SXTAB T1 |
| != 1111 | 1 | 01 | UXTAB16 T1 |
| 1111 | 1 | 10 | UXTB T2 |

| | | | |
|---|---|---|---|
| 1111 | 1 | 00 | UXTH T2 |
| 1111 | 0 | 10 | SXTB T2 |
| 1111 | 0 | 00 | SXTH T2 |
| != 1111 | 1 | 10 | UXTAB T1 |
| != 1111 | 0 | 01 | SXTAB16 T1 |

### C2.2.4.2 Parallel add-subtract

This section describes the encoding of the Parallel add-subtract. This section is decoded from Data-processing (register).




**This table shows the decode field values and the associated subgroups:**

| H | S | U | op1 | Subgroup |
|---|---|---|---|---|
| 1 | 1 | 1 | 110 | UNALLOCATED |
| - | - | - | 111 | UNALLOCATED |
| 1 | 1 | 1 | 101 | UNALLOCATED |
| 1 | 1 | 0 | 110 | UNALLOCATED |
| 1 | 1 | 1 | 100 | UNALLOCATED |
| 1 | 1 | 0 | 101 | UNALLOCATED |
| 1 | 1 | 1 | 001 | UNALLOCATED |
| 1 | 1 | 0 | 010 | UNALLOCATED |
| 1 | 1 | 1 | 010 | UNALLOCATED |
| 1 | 1 | 0 | 100 | UNALLOCATED |
| 1 | 1 | 0 | 000 | UNALLOCATED |
| 1 | 1 | 1 | 000 | UNALLOCATED |
| 1 | 1 | 0 | 001 | UNALLOCATED |

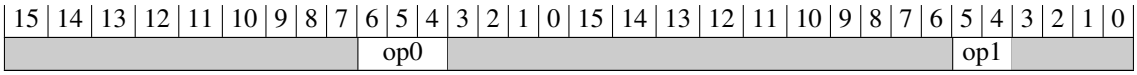
**This table shows the decode field values and the associated instrutions:**

| H | S | U | op1 | Instruction |
|---|---|---|---|---|
| 0 | 0 | 1 | 110 | USAX T1 |
| 1 | 0 | 0 | 101 | SHSUB16 T1 |
| 0 | 1 | 0 | 010 | QASX T1 |
| 0 | 0 | 1 | 001 | UADD16 T1 |
| 0 | 0 | 1 | 000 | UADD8 T1 |
| 1 | 0 | 1 | 010 | UHASX T1 |
| 0 | 0 | 1 | 101 | USUB16 T1 |
| 0 | 0 | 0 | 101 | SSUB16 T1 |
| 0 | 0 | 1 | 010 | UASX T1 |
| 0 | 0 | 1 | 100 | USUB8 T1 |
| 0 | 0 | 0 | 000 | SADD8 T1 |
| 0 | 0 | 0 | 100 | SSUB8 T1 |

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 1 | 001 | UHADD16 T1 |
| 1 | 0 | 0 | 000 | SHADD8 T1 |
| 0 | 1 | 1 | 000 | UQADD8 T1 |
| 1 | 0 | 0 | 010 | SHASX T1 |
| 0 | 1 | 0 | 000 | QADD8 T1 |
| 0 | 0 | 0 | 110 | SSAX T1 |
| 0 | 1 | 0 | 001 | QADD16 T1 |
| 0 | 0 | 0 | 010 | SASX T1 |
| 1 | 0 | 1 | 100 | UHSUB8 T1 |
| 0 | 1 | 0 | 100 | QSUB8 T1 |
| 1 | 0 | 1 | 101 | UHSUB16 T1 |
| 0 | 1 | 0 | 110 | QSAX T1 |
| 0 | 1 | 0 | 101 | QSUB16 T1 |
| 0 | 0 | 0 | 001 | SADD16 T1 |
| 1 | 0 | 0 | 001 | SHADD16 T1 |
| 0 | 1 | 1 | 001 | UQADD16 T1 |
| 1 | 0 | 1 | 110 | UHSAX T1 |
| 1 | 0 | 0 | 100 | SHSUB8 T1 |
| 0 | 1 | 1 | 010 | UQASX T1 |
| 1 | 0 | 0 | 110 | SHSAX T1 |
| 1 | 0 | 1 | 000 | UHADD8 T1 |
| 0 | 1 | 1 | 101 | UQSUB16 T1 |
| 0 | 1 | 1 | 100 | UQSUB8 T1 |
| 0 | 1 | 1 | 110 | UQSAX T1 |

### C2.2.4.3 Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers). This section is decoded from Data-processing (register).




**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|
| 011 | 1x | UNALLOCATED |
| 1xx | - | UNALLOCATED |
| 010 | 01 | UNALLOCATED |
| 010 | 1x | UNALLOCATED |
| 011 | 01 | UNALLOCATED |

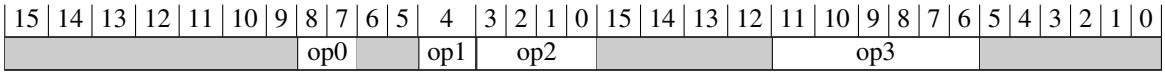
**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | Instruction |
|---|---|---|
| 000 | 01 | QDADD T1 |

| | | |
|---|---|---|
| 000 | 10 | QSUB T1 |
| 001 | 01 | REV16 T2 |
| 010 | 00 | SEL T1 |
| 011 | 00 | CLZ T1 |
| 001 | 00 | REV T2 |
| 001 | 11 | REVSH T2 |
| 001 | 10 | RBIT T1 |
| 000 | 00 | QADD T1 |
| 000 | 11 | QDSUB T1 |

### C2.2.5 Load/store single

This section describes the encoding of the Load/store single. This section is decoded from 32-bit T32 instruction encoding.

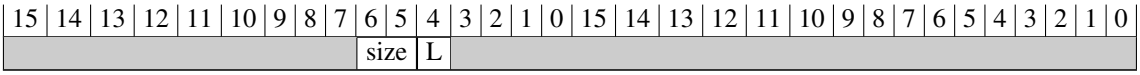



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | op2 | op3 | Subgroup |
|---|---|---|---|---|
| 00 | - | != 1111 | 11x1xx | Load/store, unsigned (immediate, pre-indexed) |
| 10 | 1 | != 1111 | 10x0xx | UNALLOCATED |
| 10 | 1 | != 1111 | 001xxx | UNALLOCATED |
| 10 | 1 | != 1111 | 01xxxx | UNALLOCATED |
| 00 | - | != 1111 | 1110xx | Load/store, unsigned (unprivileged) |
| 10 | 1 | != 1111 | 00001x | UNALLOCATED |
| 10 | 1 | != 1111 | 0001xx | UNALLOCATED |
| 00 | - | != 1111 | 001xxx | UNALLOCATED |
| 00 | - | != 1111 | 01xxxx | UNALLOCATED |
| 00 | - | != 1111 | 10x0xx | UNALLOCATED |
| 10 | 1 | != 1111 | 000001 | UNALLOCATED |
| 00 | - | != 1111 | 000001 | UNALLOCATED |
| 00 | - | != 1111 | 00001x | UNALLOCATED |
| 00 | - | != 1111 | 0001xx | UNALLOCATED |
| 10 | 1 | != 1111 | 000000 | Load/store, signed (register offset) |
| 00 | - | != 1111 | 1100xx | Load/store, unsigned (negative immediate) |
| 1x | 1 | 1111 | - | Load, signed (literal) |
| 10 | 1 | != 1111 | 11x1xx | Load/store, signed (immediate, pre-indexed) |
| 01 | - | != 1111 | - | Load/store, unsigned (positive immediate) |
| 11 | 1 | != 1111 | - | Load/store, signed (positive immediate) |
| 00 | - | != 1111 | 000000 | Load/store, unsigned (register offset) |
| 10 | 1 | != 1111 | 1110xx | Load/store, signed (unprivileged) |
| 0x | - | 1111 | - | Load, unsigned (literal) |
| 10 | 1 | != 1111 | 10x1xx | Load/store, signed (immediate, post-indexed) |
| 00 | - | != 1111 | 10x1xx | Load/store, unsigned (immediate, post-indexed) |
| 10 | 1 | != 1111 | 1100xx | Load/store, signed (negative immediate) |

### C2.2.5.1 Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

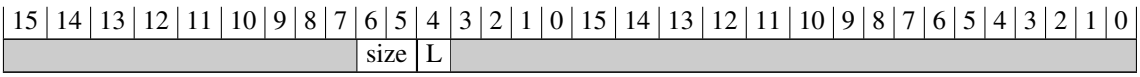
| L | size | Subgroup |
|---|---|---|
| - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | size | Instruction |
|---|---|---|
| Alias | | PUSH (single register) T4 |
| 1 | 00 | LDRB (immediate) T3 |
| Alias | | POP (single register) T4 |
| 0 | 10 | STR (immediate) T4 |
| 0 | 01 | STRH (immediate) T3 |
| 1 | 01 | LDRH (immediate) T3 |
| 0 | 00 | STRB (immediate) T3 |
| 1 | 10 | LDR (immediate) T4 |

### C2.2.5.2 Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

| L | size | Subgroup |
|---|---|---|
| - | 11 | UNALLOCATED |

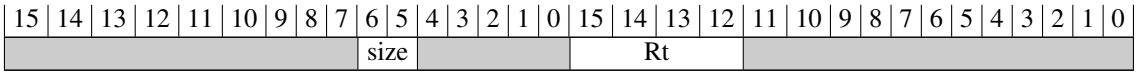
**This table shows the decode field values and the associated instrutions:**

| L | size | Instruction |
|---|---|---|

| | | |
|---|---|---|
| 1 | 10 | LDRT T1 |
| 1 | 01 | LDRHT T1 |
| 1 | 00 | LDRBT T1 |
| 0 | 10 | STRT T1 |
| 0 | 00 | STRBT T1 |
| 0 | 01 | STRHT T1 |

### C2.2.5.3 Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

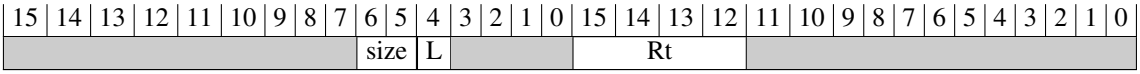
| Rt | size | Subgroup |
|---|---|---|
| - | 1x | UNALLOCATED |
| 1111 | 01 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| Rt | size | Instruction |
|---|---|---|
| != 1111 | 01 | LDRSH (register) T2 |
| != 1111 | 00 | LDRSB (register) T2 |
| 1111 | 00 | PLI (register) T1 |

### C2.2.5.4 Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

| L | Rt | size | Subgroup |
|---|---|---|---|
| - | - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | Rt | size | Instruction |
|---|---|---|---|
| Alias | | | PUSH (single register) T4 |
| 1 | != 1111 | 00 | LDRB (immediate) T3 |
| 1 | 1111 | 0x | PLD, PLDW (immediate) T2 |
| Alias | | | POP (single register) T4 |
| 0 | - | 10 | STR (immediate) T4 |
| 0 | - | 01 | STRH (immediate) T3 |
| 1 | != 1111 | 01 | LDRH (immediate) T3 |
| 0 | - | 00 | STRB (immediate) T3 |
| 1 | - | 10 | LDR (immediate) T4 |

### C2.2.5.5 Load, signed (literal)

This section describes the encoding of the Load, signed (literal). This section is decoded from Load/store single.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | size | | | | | | | Rt | | | | | | | | | | | | | | | | |

**This table shows the decode field values and the associated subgroups:**

| Rt | size | Subgroup |
|---|---|---|
| - | 1x | UNALLOCATED |
| 1111 | 01 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| Rt | size | Instruction |
|---|---|---|
| != 1111 | 01 | LDRSH (literal) T1 |
| != 1111 | 00 | LDRSB (literal) T1 |
| 1111 | 00 | PLI (immediate, literal) T3 |

### C2.2.5.6 Load/store, signed (immediate, pre-indexed)

This section describes the encoding of the Load/store, signed (immediate, pre-indexed). This section is decoded from Load/store single.

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

**This table shows the decode field values and the associated subgroups:**

| size | Subgroup |
|---|---|
| 1x | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| size | Instruction |
|---|---|
| 01 | LDRSH (immediate) T2 |
| 00 | LDRSB (immediate) T2 |

### C2.2.5.7 Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate). This section is decoded from Load/store single.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | size | | L | | | | | Rt | | | | | | | | | | | | | | | | |

**This table shows the decode field values and the associated instrutions:**

| L | Rt | size | Instruction |
|---|---|---|---|
| 1 | != 1111 | 00 | LDRB (immediate) T2 |
| 1 | 1111 | 0x | PLD, PLDW (immediate) T1 |
| 0 | - | 10 | STR (immediate) T3 |
| 1 | - | 10 | LDR (immediate) T3 |
| 0 | - | 01 | STRH (immediate) T2 |
| 0 | - | 00 | STRB (immediate) T2 |
| 1 | != 1111 | 01 | LDRH (immediate) T2 |

### C2.2.5.8 Load/store, signed (positive immediate)

This section describes the encoding of the Load/store, signed (positive immediate). This section is decoded from Load/store single.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | size | | | | | | | Rt | | | | | | | | | | | | | | | | |

**This table shows the decode field values and the associated subgroups:**

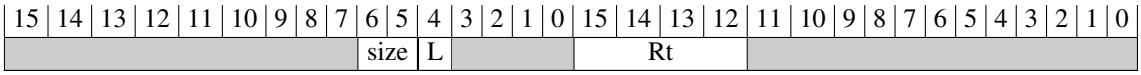
| Rt | size | Subgroup |
|---|---|---|
| 1111 | 01 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| Rt | size | Instruction |
|---|---|---|
| != 1111 | 01 | LDRSH (immediate) T1 |
| != 1111 | 00 | LDRSB (immediate) T1 |
| 1111 | 00 | PLI (immediate, literal) T1 |

### C2.2.5.9 Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

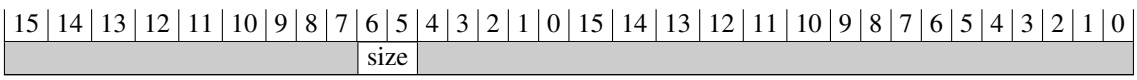
| L | Rt | size | Subgroup |
|---|---|---|---|
| - | - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | Rt | size | Instruction |
|---|---|---|---|
| 0 | - | 01 | STRH (register) T2 |
| 1 | != 1111 | 01 | LDRH (register) T2 |
| 1 | != 1111 | 00 | LDRB (register) T2 |
| 1 | 1111 | 0x | PLD, PLDW (register) T1 |
| 0 | - | 00 | STRB (register) T2 |
| 1 | - | 10 | LDR (register) T2 |
| 0 | - | 10 | STR (register) T2 |

### C2.2.5.10 Load/store, signed (unprivileged)

This section describes the encoding of the Load/store, signed (unprivileged). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

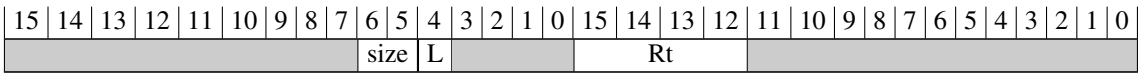
| size | Subgroup |
|---|---|
| 1x | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| size | Instruction |
|---|---|
| 00 | LDRSBT T1 |
| 01 | LDRSHT T1 |

### C2.2.5.11 Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

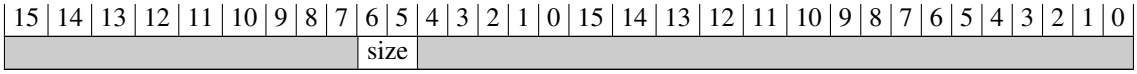
| L | Rt | size | Subgroup |
|---|---|---|---|
| - | - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | Rt | size | Instruction |
|---|---|---|---|
| 1 | - | 10 | LDR (literal) T2 |
| 1 | != 1111 | 00 | LDRB (literal) T1 |
| 1 | != 1111 | 01 | LDRH (literal) T1 |
| 1 | 1111 | 0x | PLD (literal) T1 |

### C2.2.5.12 Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

| size | Subgroup |
|---|---|

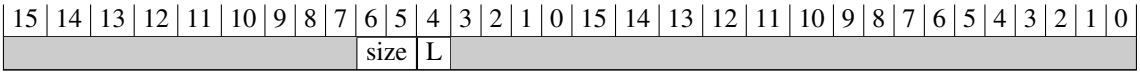
| | |
|---|---|
| 1x | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| size | Instruction |
|---|---|
| 01 | LDRSH (immediate) T2 |
| 00 | LDRSB (immediate) T2 |

### C2.2.5.13 Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed). This section is decoded from Load/store single.




**This table shows the decode field values and the associated subgroups:**

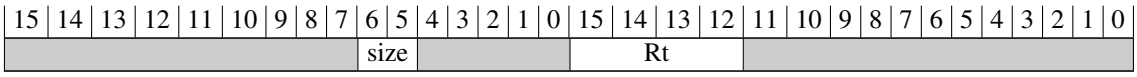
| L | size | Subgroup |
|---|---|---|
| - | 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| L | size | Instruction |
|---|---|---|
| Alias | | PUSH (single register) T4 |
| 1 | 00 | LDRB (immediate) T3 |
| Alias | | POP (single register) T4 |
| 0 | 10 | STR (immediate) T4 |
| 0 | 01 | STRH (immediate) T3 |
| 1 | 01 | LDRH (immediate) T3 |
| 0 | 00 | STRB (immediate) T3 |
| 1 | 10 | LDR (immediate) T4 |

### C2.2.5.14 Load/store, signed (negative immediate)

This section describes the encoding of the Load/store, signed (negative immediate). This section is decoded from Load/store single.



**This table shows the decode field values and the associated subgroups:**

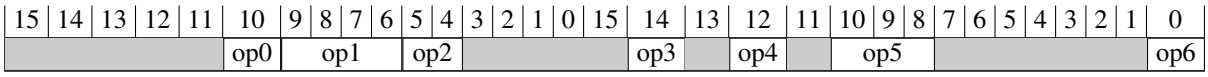
| Rt | size | Subgroup |
|---|---|---|
| - | 1x | UNALLOCATED |
| 1111 | 01 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| Rt | size | Instruction |
|---|---|---|
| != 1111 | 01 | LDRSH (immediate) T2 |
| 1111 | 00 | PLI (immediate, literal) T2 |
| != 1111 | 00 | LDRSB (immediate) T2 |

## C2.2.6 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control. This section is decoded from 32-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

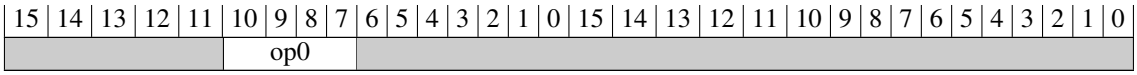
| op0 | op1 | op2 | op3 | op4 | op5 | op6 | Subgroup |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | 0 | - | 1 | Loop and branch instructions |
| 1 | 1111 | 1x | 0 | 0 | - | - | Exception generation |
| 0 | 1110 | 11 | 0 | 0 | - | - | Miscellaneous system |
| 1 | 1111 | 0x | 0 | 0 | - | - | UNALLOCATED |
| 0 | 1110 | 10 | 0 | 0 | 000 | - | Hints |
| 0 | 1110 | 10 | 0 | 0 | != 000 | - | UNALLOCATED |
| 0 | 1111 | 0x | 0 | 0 | - | - | UNALLOCATED |
| 1 | 1110 | - | 0 | 0 | - | - | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | op4 | op5 | op6 | Instruction |
|---|---|---|---|---|---|---|---|
| 0 | 1110 | 0x | 0 | 0 | - | - | MSR (register) T1 |
| - | - | - | 0 | 1 | - | - | B T4 |
| - | - | - | 1 | 1 | - | - | BL T1 |
| - | != 111x | - | 0 | 0 | - | - | B T3 |
| 0 | 1111 | 1x | 0 | 0 | - | - | MRS T1 |

### C2.2.6.1 Loop and branch instructions

This section describes the encoding of the Loop and branch instructions. This section is decoded from Branches and miscellaneous control.

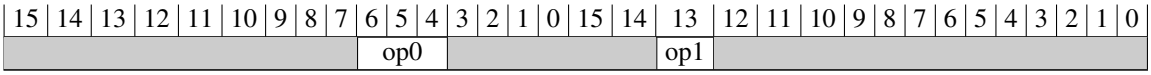



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| != 0000 | Branch future instructions |
| 0000 | Loop instructions |

#### *C2.2.6.1.1 Branch future instructions*

This section describes the encoding of the Branch future instructions. This section is decoded from Loop and branch instructions.

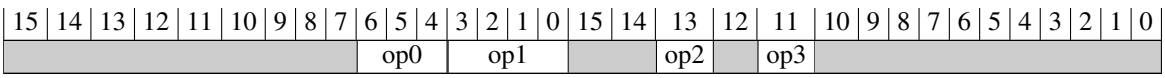



**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | Instruction |
|---|---|---|
| 10x | 1 | BF, BFX, BFL, BFLX, BFCSEL T1 |
| 110 | 1 | BF, BFX, BFL, BFLX, BFCSEL T3 |
| 0xx | 1 | BF, BFX, BFL, BFLX, BFCSEL T2 |
| 111 | 1 | BF, BFX, BFL, BFLX, BFCSEL T5 |
| - | 0 | BF, BFX, BFL, BFLX, BFCSEL T4 |

#### *C2.2.6.1.2 Loop instructions*

This section describes the encoding of the Loop instructions. This section is decoded from Loop and branch instructions.



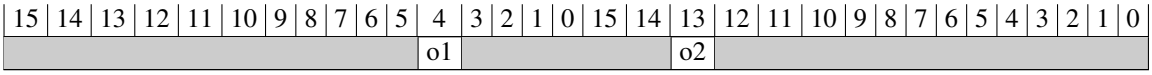

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | Instruction |
|---|---|---|---|---|
| 01x | 1111 | 0 | - | LE, LETP T2 |

| | | | | |
|---|---|---|---|---|
| 001 | 1111 | 0 | - | LE, LETP T3 |
| 000 | 1111 | 0 | - | LE, LETP T1 |
| 1xx | - | 0 | - | WLS, DLS, WLSTP, DLSTP T1 |
| 1xx | - | 1 | - | WLS, DLS, WLSTP, DLSTP T2 |
| 0xx | != 1111 | 0 | - | WLS, DLS, WLSTP, DLSTP T3 |
| 0xx | != 1111 | 1 | 1 | VCTP T1 |
| 0xx | 1111 | 1 | - | LCTP T1 |
| 0xx | != 1111 | 1 | 0 | WLS, DLS, WLSTP, DLSTP T4 |

### C2.2.6.2 Exception generation

This section describes the encoding of the Exception generation. This section is decoded from Branches and miscellaneous control.




**This table shows the decode field values and the associated subgroups:**

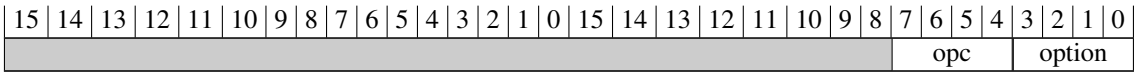
| o1 | o2 | Subgroup |
|---|---|---|
| 0 | 0 | UNALLOCATED |
| 0 | 1 | UNALLOCATED |
| 1 | 0 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| o1 | o2 | Instruction |
|---|---|---|
| 1 | 1 | UDF T2 |

### C2.2.6.3 Miscellaneous system

This section describes the encoding of the Miscellaneous system. This section is decoded from Branches and miscellaneous control.




**This table shows the decode field values and the associated subgroups:**

| opc | option | Subgroup |
|---|---|---|
| 1xxx | - | UNALLOCATED |
| 000x | - | UNALLOCATED |

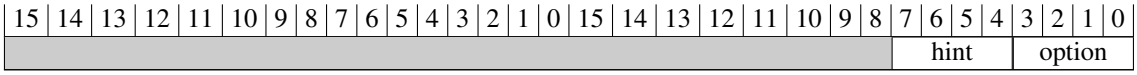
| | | |
|---|---|---|
| 0011 | - | UNALLOCATED |
| 0111 | - | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| opc | option | Instruction |
|---|---|---|
| 0100 | 0000 | SSBB T1 |
| 0101 | - | DMB T1 |
| 0010 | - | CLREX T1 |
| 0110 | - | ISB T1 |
| 0100 | != 0x00 | DSB T1 |
| 0100 | 0100 | PSSBB T1 |

### C2.2.6.4 Hints

This section describes the encoding of the Hints. This section is decoded from Branches and miscellaneous control.




**This table shows the decode field values and the associated subgroups:**

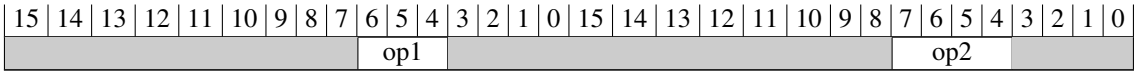
| hint | option | Subgroup |
|---|---|---|
| 1110 | - | Reserved hint, behaves as NOP |
| 10xx | - | Reserved hint, behaves as NOP |
| 110x | - | Reserved hint, behaves as NOP |
| 001x | - | Reserved hint, behaves as NOP |
| 01xx | - | Reserved hint, behaves as NOP |
| 0001 | != 0100 | Reserved hint, behaves as NOP |
| 0001 | 1xxx | Reserved hint, behaves as NOP |
| 0000 | 011x | Reserved hint, behaves as NOP |
| 0000 | 1xxx | Reserved hint, behaves as NOP |
| 0000 | 0101 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| hint | option | Instruction |
|---|---|---|
| 0001 | 0000 | ESB T1 |
| 0000 | 0100 | SEV T2 |
| 0000 | 0011 | WFI T2 |
| 0001 | 0100 | CSDB T1 |
| 1111 | - | DBG T1 |
| 0000 | 0010 | WFE T2 |
| 0000 | 0000 | NOP T2 |
| 0000 | 0001 | YIELD T2 |

### C2.2.7 Long multiply and divide

This section describes the encoding of the Long multiply and divide. This section is decoded from 32-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

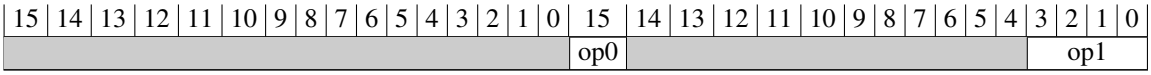
| op1 | op2 | Subgroup |
|---|---|---|
| 110 | 1xxx | UNALLOCATED |
| 111 | - | UNALLOCATED |
| 110 | 010x | UNALLOCATED |
| 110 | 0111 | UNALLOCATED |
| 110 | 0001 | UNALLOCATED |
| 110 | 001x | UNALLOCATED |
| 101 | 10xx | UNALLOCATED |
| 101 | 111x | UNALLOCATED |
| 100 | 111x | UNALLOCATED |
| 101 | 0xxx | UNALLOCATED |
| 011 | != 1111 | UNALLOCATED |
| 100 | 0001 | UNALLOCATED |
| 100 | 001x | UNALLOCATED |
| 100 | 01xx | UNALLOCATED |
| 000 | != 0000 | UNALLOCATED |
| 001 | != 1111 | UNALLOCATED |
| 010 | != 0000 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| op1 | op2 | Instruction |
|---|---|---|
| 110 | 0110 | UMAAL T1 |
| 100 | 110x | SMLALD, SMLALDX T1 |
| 110 | 0000 | UMLAL T1 |
| 100 | 0000 | SMLAL T1 |
| 100 | 10xx | SMLALBB, SMLALBT, SMLALTB, SMLALTT T1 |
| 010 | 0000 | UMULL T1 |
| 000 | 0000 | SMULL T1 |
| 011 | 1111 | UDIV T1 |
| 101 | 110x | SMLSLD, SMLSLDX T1 |
| 001 | 1111 | SDIV T1 |

### C2.2.8 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register). This section is decoded from 32-bit T32 instruction encoding.

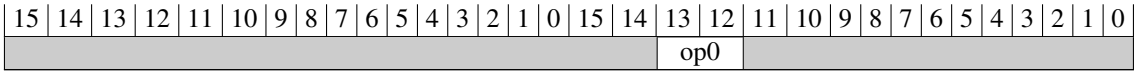



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|
| 1 | - | Conditional select instructions |
| 0 | 1111 | 64bit shift immediate instructions |
| 0 | != 1111 | 64bit shift register instructions |

#### C2.2.8.1 Conditional select instructions

This section describes the encoding of the Conditional select instructions. This section is decoded from Data-processing (shifted register).

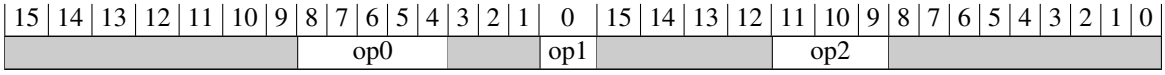



**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 01 | CSINC T1 |
| Alias | CNEG T1 |
| Alias | CSET T1 |
| Alias | CINV T1 |
| 11 | CSNEG T1 |
| 00 | CSEL T1 |
| 10 | CSINV T1 |
| Alias | CINC T1 |
| Alias | CSETM T1 |

#### C2.2.8.2 64bit shift immediate instructions

This section describes the encoding of the 64bit shift immediate instructions. This section is decoded from Data-processing (shifted register).

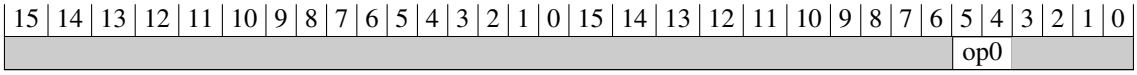


**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | op2 | Subgroup |
|---|---|---|---|
| - | 1 | != 111 | 64-bit wide-shift saturating and rounding (two general-purpose registers) |
| 00101 | - | 111 | 64-bit shift (one general-purpose register) |
| - | 0 | != 111 | 64-bit wide-shift (two general-purpose registers) |
| != 00101 | - | 111 | UNALLOCATED |

#### C2.2.8.2.1 64-bit wide-shift saturating and rounding (two general-purpose registers)

This section describes the encoding of the 64-bit wide-shift saturating and rounding (two general-purpose registers). This section is decoded from 64bit shift immediate instructions.

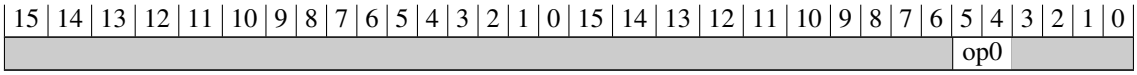



**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 11 | SQSHLL (immediate) T1 |
| 01 | URSHRL (immediate) T1 |
| 10 | SRSHRL (immediate) T1 |
| 00 | UQSHLL (immediate) T1 |

#### C2.2.8.2.2 64-bit shift (one general-purpose register)

This section describes the encoding of the 64-bit shift (one general-purpose register). This section is decoded from 64bit shift immediate instructions.

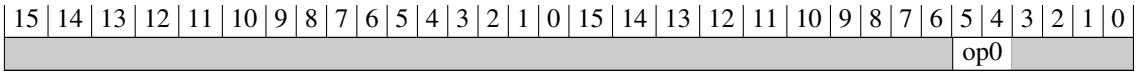



**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 11 | SQSHL (immediate) T1 |
| 10 | SRSHR (immediate) T1 |
| 00 | UQSHL (immediate) T1 |
| 01 | URSHR (immediate) T1 |

#### C2.2.8.2.3 64-bit wide-shift (two general-purpose registers)

This section describes the encoding of the 64-bit wide-shift (two general-purpose registers). This section is decoded from 64bit shift immediate instructions.




**This table shows the decode field values and the associated subgroups:**

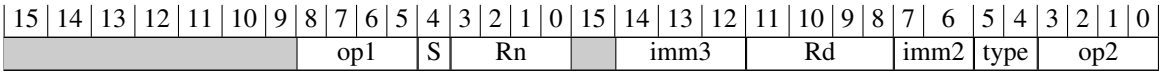
| op0 | Subgroup |
|---|---|
| 11 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 00 | LSLL (immediate) T1 |
| 10 | ASRL (immediate) T1 |
| 01 | LSRL (immediate) T1 |

### C2.2.8.3 64bit shift register instructions

This section describes the encoding of the 64bit shift register instructions. This section is decoded from Data-processing (shifted register).




**This table shows the decode field values and the associated subgroups:**

| Rd | Rn | S | imm3:imm2:type | op1 | op2 | Subgroup |
|---|---|---|---|---|---|---|
| - | - | - | - | 0111 | - | UNALLOCATED |
| - | - | - | - | 1001 | - | UNALLOCATED |
| - | - | - | - | 1100 | - | UNALLOCATED |
| - | - | - | - | 1111 | - | UNALLOCATED |
| - | - | - | - | 0101 | - | UNALLOCATED |
| - | - | 0 | xxxxx01 | 0110 | - | UNALLOCATED |
| - | - | 0 | xxxxx11 | 0110 | - | UNALLOCATED |

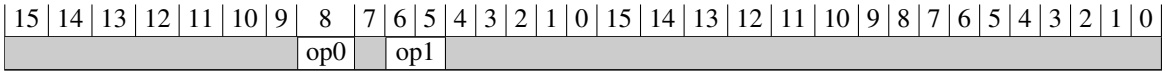
**This table shows the decode field values and the associated instrutions:**

| Rd | Rn | S | imm3:imm2:type | op1 | op2 | Instruction |
|---|---|---|---|---|---|---|
| Alias | | | | | | RORS (immediate) T3 |

| | | | | | | |
|---|---|---|---|---|---|---|
| - | - | - | - | 1110 | - | RSB (register) T1 |
| - | != 1111 | 0 | - | 0011 | - | ORN (register) T1 |
| - | != 1111 | 1 | - | 0010 | - | ORRS (register) |
| - | 1111 | 0 | - | 0011 | - | MVN (register) T2 |
| Alias | | | | | | LSLS (immediate) T3 |
| 111x | - | 1 | xxxxx00 | 0010 | 1101 | UQRSHL (register) T1 |
| != 1111 | - | 1 | 0000011 | 0100 | - | EORS extend (register) |
| != 1111 | != 1101 | 1 | - | 1101 | - | SUBS (register) |
| - | != 1111 | 1 | - | 0011 | - | ORNS (register) |
| - | != 1101 | 0 | - | 1101 | - | SUB (register) T2 |
| 1111 | - | 1 | - | 1000 | - | CMN (register) T2 |
| 1111 | - | 1 | - | 1101 | - | CMP (register) T3 |
| != 111x | xxx0 | 1 | xxxxx00 | 0010 | 1101 | LSLL (register) T1 |
| - | - | 0 | xxxxxx0 | 0110 | - | PKHBT, PKHTB T1 |
| != 111x | xxx0 | 1 | xxxxx10 | 0010 | 1101 | ASRL (register) T1 |
| 111x | - | 1 | xxxxx10 | 0010 | 1101 | SQRSHR (register) T1 |
| - | - | - | - | 1011 | - | SBC (register) T2 |
| - | 1101 | 0 | - | 1101 | - | SUB (SP minus register) T1 |
| - | 1111 | 0 | - | 0010 | != 1101 | MOV (register) T3 |
| - | 1101 | 0 | - | 1000 | - | ADD (SP plus register) T3 |
| Alias | | | | | | RRX T3 |
| Alias | | | | | | ASRS (immediate) T3 |
| - | 1111 | 1 | - | 0011 | - | MVNS (register) |
| Alias | | | | | | ROR (immediate) T3 |
| != 111x | xxx1 | 1 | xxxxx00 | 0010 | 1101 | UQRSHLL (register) T1 |
| != 1111 | 1101 | 1 | - | 1000 | - | ADDS (SP plus register) |
| - | != 1111 | 0 | - | 0010 | != 1101 | ORR (register) T2 |
| - | != 1101 | 0 | - | 1000 | - | ADD (register) T3 |
| != 1111 | 1101 | 1 | - | 1101 | - | SUBS (SP minus register) |
| - | - | - | - | 1010 | - | ADC (register) T2 |
| - | - | 0 | - | 0100 | - | EOR (register) T2 |
| - | - | 0 | - | 0000 | - | AND (register) T2 |
| Alias | | | | | | ASR (immediate) T3 |
| 1111 | - | 1 | != 0000011 | 0000 | - | TST (register) T2 |
| Alias | | | | | | LSR (immediate) T3 |
| != 111x | xxx1 | 1 | xxxxx10 | 0010 | 1101 | SQRSHRL (register) T1 |
| Alias | | | | | | LSL (immediate) T3 |
| != 1111 | != 1101 | 1 | - | 1000 | - | ADDS (register) |
| - | 1111 | 1 | - | 0010 | - | MOVS (register) |
| 1111 | - | 1 | 0000011 | 0100 | - | TEQ extend (register) |
| != 1111 | - | 1 | != 0000011 | 0100 | - | EORS (register) |
| != 1111 | - | 1 | 0000011 | 0000 | - | ANDS extend (register) |
| != 1111 | - | 1 | != 0000011 | 0000 | - | ANDS (register) |
| 1111 | - | 1 | != 0000011 | 0100 | - | TEQ (register) T1 |
| Alias | | | | | | RRXS T3 |
| Alias | | | | | | LSRS (immediate) T3 |
| 1111 | - | 1 | 0000011 | 0000 | - | TST extend (register) |
| - | - | - | - | 0001 | - | BIC (register) T2 |

### C2.2.9 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate). This section is decoded from 32-bit T32 instruction encoding.

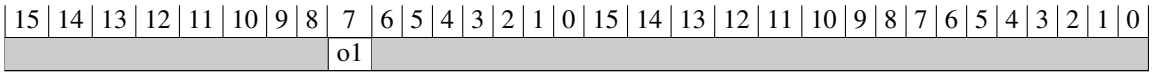



**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | Subgroup |
|---|---|---|
| 0 | 10 | Move Wide (16-bit immediate) |
| 1 | - | Saturate, bitfield |
| 0 | 0x | Data-processing (simple immediate) |
| 0 | 11 | UNALLOCATED |

#### C2.2.9.1 Move Wide (16-bit immediate)

This section describes the encoding of the Move Wide (16-bit immediate). This section is decoded from Data-processing (plain binary immediate).

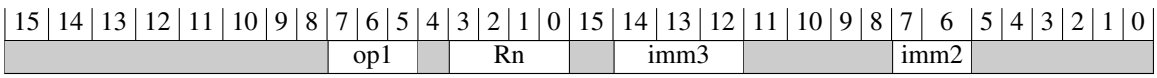



**This table shows the decode field values and the associated instrutions:**

| o1 | Instruction |
|---|---|
| 1 | MOVT T1 |
| 0 | MOV (immediate) T3 |

#### C2.2.9.2 Saturate, bitfield

This section describes the encoding of the Saturate, bitfield. This section is decoded from Data-processing (plain binary immediate).




**This table shows the decode field values and the associated subgroups:**

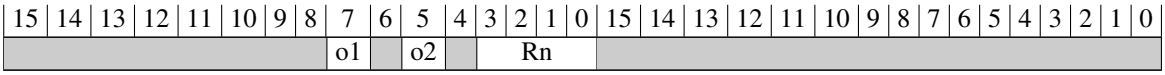
| Rn | imm3:imm2 | op1 | Subgroup |
|---|---|---|---|
| - | - | 111 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| Rn | imm3:imm2 | op1 | Instruction |
|---|---|---|---|
| - | - | 100 | USAT T1 |
| - | != 00000 | 001 | SSAT |
| - | - | 010 | SBFX T1 |
| - | != 00000 | 101 | USAT |
| - | 00000 | 001 | SSAT16 T1 |
| - | 00000 | 101 | USAT16 T1 |
| - | - | 110 | UBFX T1 |
| != 1111 | - | 011 | BFI T1 |
| - | - | 000 | SSAT T1 |
| 1111 | - | 011 | BFC T1 |

### C2.2.9.3 Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate). This section is decoded from Data-processing (plain binary immediate).




**This table shows the decode field values and the associated subgroups:**

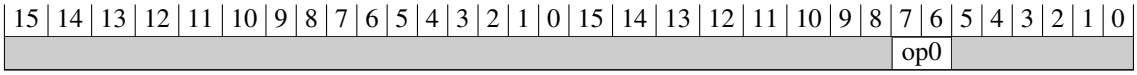
| Rn | o1 | o2 | Subgroup |
|---|---|---|---|
| - | 0 | 1 | UNALLOCATED |
| - | 1 | 0 | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| Rn | o1 | o2 | Instruction |
|---|---|---|---|
| != 11x1 | 1 | 1 | SUB (immediate) T4 |
| Alias | | | ADD (immediate, to PC) T3 |
| 1101 | 0 | 0 | ADD (SP plus immediate) T4 |
| 1111 | 0 | 0 | ADR T3 |
| 1111 | 1 | 1 | ADR T2 |
| Alias | | | SUB (immediate, from PC) T2 |
| 1101 | 1 | 1 | SUB (SP minus immediate) T3 |
| != 11x1 | 0 | 0 | ADD (immediate) T4 |

### C2.2.10 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference. This section is decoded from 32-bit T32 instruction encoding.

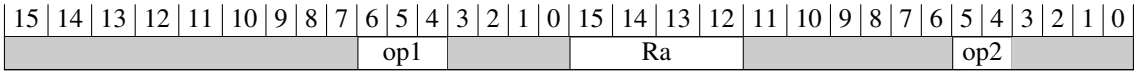



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| 00 | Multiply and absolute difference |
| 01 | UNALLOCATED |
| 1x | UNALLOCATED |

#### C2.2.10.1 Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference. This section is decoded from Multiply, multiply accumulate, and absolute difference.




**This table shows the decode field values and the associated subgroups:**

| Ra | op1 | op2 | Subgroup |
|---|---|---|---|
| - | 111 | 1x | UNALLOCATED |
| - | 100 | 1x | UNALLOCATED |
| - | 101 | 1x | UNALLOCATED |
| - | 110 | 1x | UNALLOCATED |
| - | 111 | 01 | UNALLOCATED |
| - | 000 | 1x | UNALLOCATED |
| - | 010 | 1x | UNALLOCATED |
| - | 011 | 1x | UNALLOCATED |

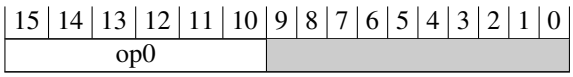
**This table shows the decode field values and the associated instrutions:**

| Ra | op1 | op2 | Instruction |
|---|---|---|---|
| 1111 | 100 | 0x | SMUSD, SMUSDX T1 |
| - | 110 | 0x | SMMLS, SMMLSR T1 |
| 1111 | 111 | 00 | USAD8 T1 |
| != 1111 | 011 | 0x | SMLAWB, SMLAWT T1 |
| != 1111 | 101 | 0x | SMMLA, SMMLAR T1 |
| != 1111 | 010 | 0x | SMLAD, SMLADX T1 |
| 1111 | 001 | - | SMULBB, SMULBT, SMULTB, SMULTT T1 |

| | | | |
|---|---|---|---|
| != 1111 | 000 | 00 | MLA T1 |
| 1111 | 000 | 00 | MUL T2 |
| 1111 | 010 | 0x | SMUAD, SMUADX T1 |
| - | 000 | 01 | MLS T1 |
| 1111 | 101 | 0x | SMMUL, SMMULR T1 |
| 1111 | 011 | 0x | SMULWB, SMULWT T1 |
| != 1111 | 100 | 0x | SMLSD, SMLSDX T1 |
| != 1111 | 111 | 00 | USADA8 T1 |
| != 1111 | 001 | - | SMLABB, SMLABT, SMLATB, SMLATT T1 |

## C2.3 16-bit T32 instruction encoding

This section describes the encoding of the 16-bit T32 instruction encoding. This section is decoded from Top level T32 instruction set encoding.




**This table shows the decode field values and the associated subgroups:**

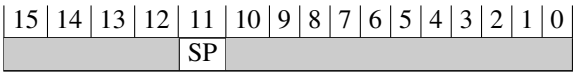
| op0 | Subgroup |
|---|---|
| 1010xx | Add PC/SP (immediate) |
| 010000 | Data-processing (two low registers) |
| 0101xx | Load/store (register offset) |
| 1100xx | Load/store multiple |
| 1101xx | Conditional branch, and Supervisor Call |
| 1001xx | Load/store (SP-relative) |
| 011xxx | Load/store word/byte (immediate offset) |
| 010001 | Special data instructions and branch and exchange |
| 1000xx | Load/store halfword (immediate offset) |
| 00xxxx | Shift (immediate), add, subtract, move, and compare |
| 1011xx | Miscellaneous 16-bit instructions |

**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| 01001x | LDR (literal) T1 |

### C2.3.1 Add PC/SP (immediate)

This section describes the encoding of the Add PC/SP (immediate). This section is decoded from 16-bit T32 instruction encoding.

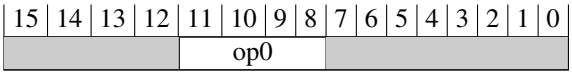



**This table shows the decode field values and the associated instrutions:**

| SP | Instruction |
|---|---|
| 0 | ADR T1 |
| Alias | ADD (immediate, to PC) T1 |
| 1 | ADD (SP plus immediate) T1 |

### C2.3.2 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call. This section is decoded from 16-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

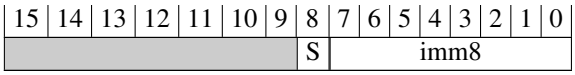
| op0 | Subgroup |
|---|---|
| 111x | Exception generation |

**This table shows the decode field values and the associated instrutions:**

| op0 | Instruction |
|---|---|
| != 111x | B T1 |

#### C2.3.2.1 Exception generation

This section describes the encoding of the Exception generation. This section is decoded from Conditional branch, and Supervisor Call.

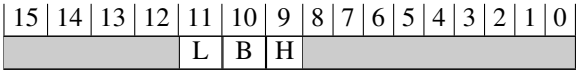



**This table shows the decode field values and the associated instrutions:**

| S | imm8 | Instruction |
|---|---|---|
| 1 | - | SVC T1 |
| 0 | - | UDF T1 |

### C2.3.3 Load/store (register offset)

This section describes the encoding of the Load/store (register offset). This section is decoded from 16-bit T32 instruction encoding.

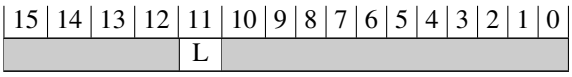


**This table shows the decode field values and the associated instrutions:**

| B | H | L | Instruction |
|---|---|---|---|
| 0 | 1 | 0 | STRH (register) T1 |
| 1 | 1 | 0 | LDRSB (register) T1 |
| 0 | 1 | 1 | LDRH (register) T1 |
| 1 | 0 | 1 | LDRB (register) T1 |
| 1 | 0 | 0 | STRB (register) T1 |
| 1 | 1 | 1 | LDRSH (register) T1 |
| 0 | 0 | 1 | LDR (register) T1 |
| 0 | 0 | 0 | STR (register) T1 |

### C2.3.4 Load/store (SP-relative)

This section describes the encoding of the Load/store (SP-relative). This section is decoded from 16-bit T32 instruction encoding.

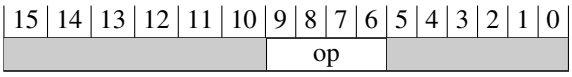



**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| 1 | LDR (immediate) T2 |
| 0 | STR (immediate) T2 |

### C2.3.5 Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers). This section is decoded from 16-bit T32 instruction encoding.




**This table shows the decode field values and the associated instrutions:**

| op | Instruction |
|---|---|
| Alias | ASR (register) T1 |
| Alias | LSR (register) T1 |
| Alias | ROR (register) T1 |
| Alias | ASRS (register) T1 |
| 0110 | SBC (register) T1 |

| | |
|---|---|
| 1011 | CMN (register) T1 |
| 1000 | TST (register) T1 |
| 1100 | ORR (register) T1 |
| Alias | LSLS (register) T1 |
| 1001 | RSB (immediate) T1 |
| Alias | LSRS (register) T1 |
| 0101 | ADC (register) T1 |
| Alias | LSL (register) T1 |
| 0000 | AND (register) T1 |
| 0001 | EOR (register) T1 |
| 0010<br>0011<br>0100<br>0111 | MOV, MOVS (register-shifted register) T1 |
| 1101 | MUL T1 |
| 1010 | CMP (register) T1 |
| 1110 | BIC (register) T1 |
| Alias | RORS (register) T1 |
| 1111 | MVN (register) T1 |

### C2.3.6 Load/store multiple

This section describes the encoding of the Load/store multiple. This section is decoded from 16-bit T32 instruction encoding.

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

**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| 1 | LDM, LDMIA, LDMFD T1 |
| 0 | STM, STMIA, STMEA T1 |

### C2.3.7 Load/store word/byte (immediate offset)

This section describes the encoding of the Load/store word/byte (immediate offset). This section is decoded from 16-bit T32 instruction encoding.

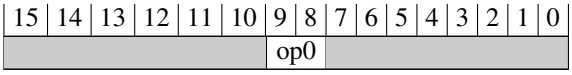
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | B | L | | | | | | | | | | | |

**This table shows the decode field values and the associated instrutions:**

| B | L | Instruction |
|---|---|---|
| 0 | 1 | LDR (immediate) T1 |
| 1 | 1 | LDRB (immediate) T1 |
| 0 | 0 | STR (immediate) T1 |
| 1 | 0 | STRB (immediate) T1 |

### C2.3.8 Special data instructions and branch and exchange

This section describes the encoding of the Special data instructions and branch and exchange. This section is decoded from 16-bit T32 instruction encoding.

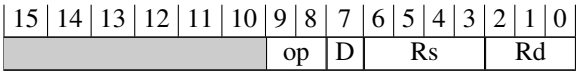



**This table shows the decode field values and the associated subgroups:**

| op0 | Subgroup |
|---|---|
| != 11 | Add, subtract, compare, move (two high registers) |
| 11 | Branch and exchange |

#### C2.3.8.1 Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers). This section is decoded from Special data instructions and branch and exchange.

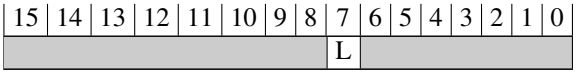



**This table shows the decode field values and the associated instrutions:**

| D:Rd | Rs | op | Instruction |
|---|---|---|---|
| - | 1101 | 00 | ADD (SP plus register) T1 |
| - | - | 10 | MOV (register) T1 |
| 1101 | != 1101 | 00 | ADD (SP plus register) T2 |
| != 1101 | != 1101 | 00 | ADD (register) T2 |
| - | - | 01 | CMP (register) T2 |

#### C2.3.8.2 Branch and exchange

This section describes the encoding of the Branch and exchange. This section is decoded from Special data instructions and branch and exchange.

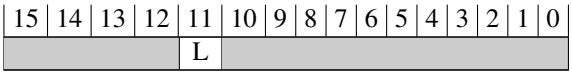



**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| 0 | BX, BXNS T1 |
| 1 | BLX, BLXNS T1 |

## C2.3.9 Load/store halfword (immediate offset)

This section describes the encoding of the Load/store halfword (immediate offset). This section is decoded from 16-bit T32 instruction encoding.

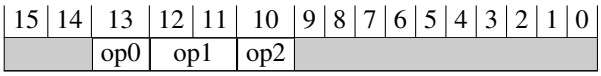



**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| 0 | STRH (immediate) T1 |
| 1 | LDRH (immediate) T1 |

## C2.3.10 Shift (immediate), add, subtract, move, and compare

This section describes the encoding of the Shift (immediate), add, subtract, move, and compare. This section is decoded from 16-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

| op0 | op1 | op2 | Subgroup |
|---|---|---|---|
| 1 | - | - | Add, subtract, compare, move (one low register and immediate) |
| 0 | 11 | 0 | Add, subtract (three low registers) |
| 0 | 11 | 1 | Add, subtract (two low registers and immediate) |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | Instruction |
|---|---|---|---|
| Alias | | | LSL (immediate) T2 |
| Alias | | | LSLS (immediate) T2 |
| Alias | | | ASR (immediate) T2 |
| 0 | != 11 | - | MOV (register) T2 |
| Alias | | | LSR (immediate) T2 |
| Alias | | | ASRS (immediate) T2 |
| Alias | | | LSRS (immediate) T2 |

### C2.3.10.1 Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate). This section is decoded from Shift (immediate), add, subtract, move, and compare.

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

**This table shows the decode field values and the associated instrutions:**

| op | Instruction |
|---|---|
| 10 | ADD (immediate) T2 |
| 11 | SUB (immediate) T2 |
| 01 | CMP (immediate) T1 |
| 00 | MOV (immediate) T1 |

### C2.3.10.2 Add, subtract (three low registers)

This section describes the encoding of the Add, subtract (three low registers). This section is decoded from Shift (immediate), add, subtract, move, and compare.

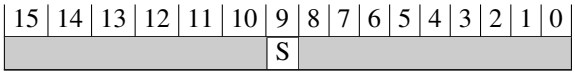
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | S | | | | | | | | | |

**This table shows the decode field values and the associated instrutions:**

| S | Instruction |
|---|---|
| 1 | SUB (register) T1 |
| 0 | ADD (register) T1 |

### C2.3.10.3 Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate). This section is decoded from Shift (immediate), add, subtract, move, and compare.

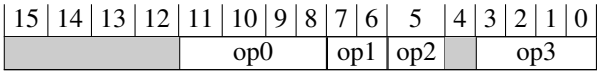



**This table shows the decode field values and the associated instrutions:**

| S | Instruction |
|---|---|
| 0 | ADD (immediate) T1 |
| 1 | SUB (immediate) T1 |

## C2.3.11 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions. This section is decoded from 16-bit T32 instruction encoding.




**This table shows the decode field values and the associated subgroups:**

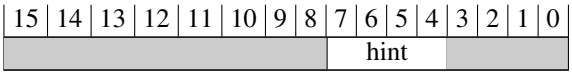
| op0 | op1 | op2 | op3 | Subgroup |
|---|---|---|---|---|
| 1111 | - | - | 0000 | Hints |
| 1010 | != 10 | - | - | Reverse bytes |
| 0000 | - | - | - | Adjust SP (immediate) |
| 1000 | - | - | - | UNALLOCATED |
| 0111 | - | - | - | UNALLOCATED |
| 0010 | - | - | - | Extend |
| 1010 | 10 | - | - | UNALLOCATED |
| x10x | - | - | - | Push and Pop |
| 0110 | 00 | - | - | UNALLOCATED |
| 0110 | 01 | 0 | - | UNALLOCATED |
| 0110 | 1x | - | - | UNALLOCATED |

**This table shows the decode field values and the associated instrutions:**

| op0 | op1 | op2 | op3 | Instruction |
|---|---|---|---|---|
| 1110 | - | - | - | BKPT T1 |
| 0110 | 01 | 1 | - | CPS T1 |
| 1111 | - | - | != 0000 | IT T1 |
| x0x1 | - | - | - | CBNZ, CBZ T1 |

### C2.3.11.1 Hints

This section describes the encoding of the Hints. This section is decoded from Miscellaneous 16-bit instructions.




**This table shows the decode field values and the associated subgroups:**

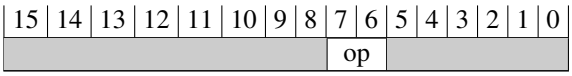
| hint | Subgroup |
|---|---|
| 011x | Reserved hint, behaves as NOP |
| 1xxx | Reserved hint, behaves as NOP |
| 0101 | Reserved hint, behaves as NOP |

**This table shows the decode field values and the associated instrutions:**

| hint | Instruction |
|---|---|
| 0011 | WFI T1 |
| 0001 | YIELD T1 |
| 0100 | SEV T1 |
| 0000 | NOP T1 |
| 0010 | WFE T1 |

### C2.3.11.2 Reverse bytes

This section describes the encoding of the Reverse bytes. This section is decoded from Miscellaneous 16-bit instructions.

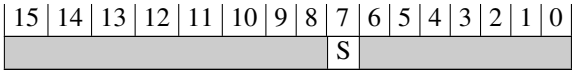



**This table shows the decode field values and the associated instrutions:**

| op | Instruction |
|---|---|
| 01 | REV16 T1 |
| 11 | REVSH T1 |
| 00 | REV T1 |

### C2.3.11.3 Adjust SP (immediate)

This section describes the encoding of the Adjust SP (immediate). This section is decoded from Miscellaneous 16-bit instructions.

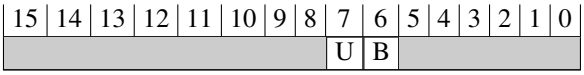



**This table shows the decode field values and the associated instrutions:**

| S | Instruction |
|---|---|
| 1 | SUB (SP minus immediate) T1 |
| 0 | ADD (SP plus immediate) T2 |

### C2.3.11.4 Extend

This section describes the encoding of the Extend. This section is decoded from Miscellaneous 16-bit instructions.

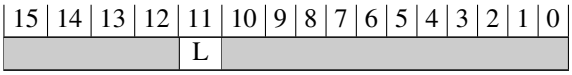



**This table shows the decode field values and the associated instrutions:**

| B | U | Instruction |
|---|---|---|
| 1 | 1 | UXTB T1 |
| 0 | 1 | UXTH T1 |
| 0 | 0 | SXTH T1 |
| 1 | 0 | SXTB T1 |

### C2.3.11.5 Push and Pop

This section describes the encoding of the Push and Pop. This section is decoded from Miscellaneous 16-bit instructions.




**This table shows the decode field values and the associated instrutions:**

| L | Instruction |
|---|---|
| Alias | POP (multiple registers) T3 |
| Alias | PUSH (multiple registers) T2 |
| 0 | STMDB, STMFD T2 |

| | |
|---|---|
| 1 | LDM, LDMIA, LDMFD T3 |

## C2.4 Alphabetical list of instructions

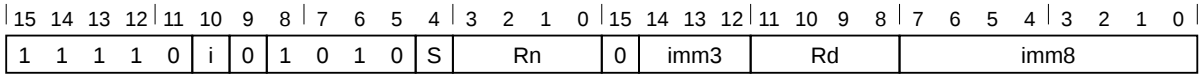
Every Armv8-M instruction is listed in this section. See Chapter C1 *Instruction Set Overview* on page 420 for the format of the instruction descriptions.

## C2.4.1 ADC (immediate)

Add with Carry (immediate). Add with Carry (immediate) adds an immediate value and the carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**ADC variant**

**Applies when `S == 0`.**

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**ADCS variant**

**Applies when `S == 1`.**

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
3 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

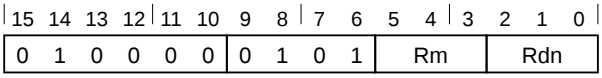
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
4     R[d] = result;
5     if setflags then
6         APSR.N = result<31>;
7         APSR.Z = IsZeroBit(result);
8         APSR.C = carry;
9         APSR.V = overflow;
```

## C2.4.2 ADC (register)

Add with Carry (register). Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




#### T1 variant

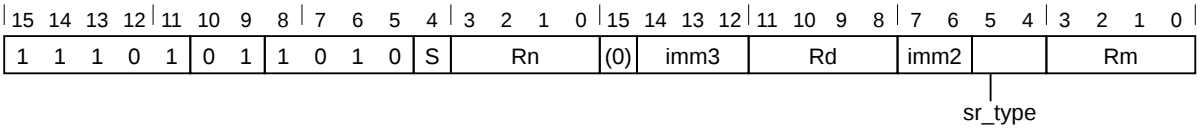
```
ADC<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
ADCS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### ADC, rotate right with extend variant

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### ADC, shift or rotate by value variant

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ADC<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

#### ADCS, rotate right with extend variant

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

#### ADCS, shift or rotate by value variant

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ADCS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

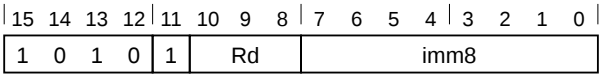
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

### C2.4.3 ADD (SP plus immediate)

Add to SP (immediate). ADD (SP plus immediate) adds an immediate value to the SP value, and writes the result to the destination register.

#### T1

*Armv8-M*




**T1 variant**

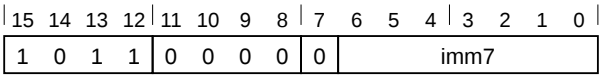
```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

#### Decode for this encoding

```
1  d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

#### T2

*Armv8-M*




**T2 variant**

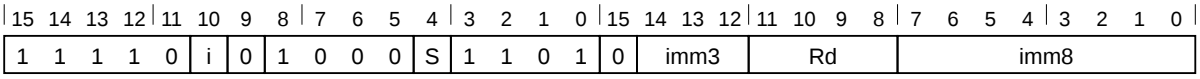
```
ADD{<c>}{<q>} {SP,} SP, #<imm7>
```

#### Decode for this encoding

```
1  d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

#### T3

*Armv8-M Main Extension only*




**ADD variant**

**Applies when `S == 0`.**

```
ADD{<c>}.W {<Rd>,} SP, #<const>
  // <Rd>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, #<const>
```

**ADDS variant**

**Applies when `S == 1 && Rd != 1111`.**

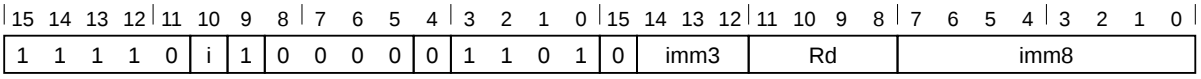
```
ADDS{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### Decode for this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && S == '0' then UNPREDICTABLE;
```

### T4

*Armv8-M Main Extension only*




#### T4 variant

```
ADD{<c>}{<q>} {<Rd>,} SP, #<imm12>
  // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} SP, #<imm12>
  // <imm12> can be represented in T1, T2, or T3
```

## Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <imm7> | Is an unsigned immediate, a multiple of 4 in the range 0 to 508, encoded in the "imm7" field as <imm7>/4. |
| <Rd> | For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.<br>For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. |
| <imm8> | Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4. |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

## Operation for all encodings

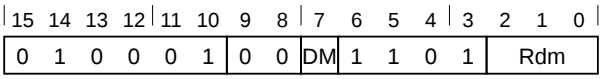
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    RSPCheck[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

### C2.4.4 ADD (SP plus register)

Add to SP (register). ADD (SP plus register) adds an optionally-shifted register value to the SP value, and writes the result to the destination register.

#### T1

*Armv8-M*




**T1 variant**

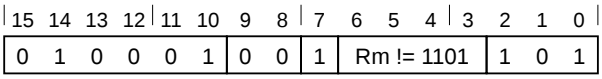
```
ADD{<c>}{<q>} {<Rdm>,} SP, <Rdm>
```

#### Decode for this encoding

```
d = UInt(DM:Rdm);  m = UInt(DM:Rdm);  setflags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

#### T2

*Armv8-M*




**T2 variant**

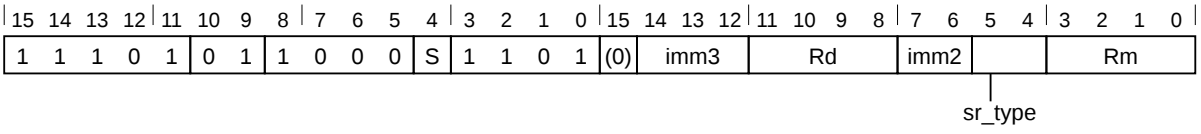
```
ADD{<c>}{<q>} {SP,} SP, <Rm>
```

#### Decode for this encoding

```
if Rm == '1101' then SEE "encoding T1";
d = 13;  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

#### T3

*Armv8-M Main Extension only*




**ADD, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

**ADD, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ADD{<c>}.W {<Rd>,} SP, <Rm>
  // <Rd>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

### ADDS, rotate right with extend variant

**Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11`.**

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

### ADDS, shift or rotate by value variant

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111`.**

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "CMN (register)";
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
4  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5  if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
6  if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the general-purpose destination and second source register, encoded in the "Rdm" field. If omitted, this register is the SP. Arm deprecates using the PC as the destination register, but if the PC is used, the instruction is a simple branch to the address calculated by the operation. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. |
| <Rm> | For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.<br>For encoding T3: is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when `sr_type = 00`<br>LSR when `sr_type = 01`<br>ASR when `sr_type = 10`<br>ROR when `sr_type = 11` |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

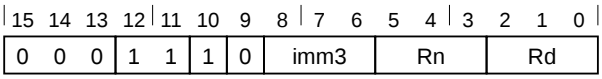
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4      (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
5      if d == 15 then
6          BranchTo(result);  // setflags is always FALSE here
7      else
8          RSPCheck[d] = result;
9          if setflags then
10             APSR.N = result<31>;
11             APSR.Z = IsZeroBit(result);
12             APSR.C = carry;
13             APSR.V = overflow;
```

## C2.4.5 ADD (immediate)

Add (immediate). Add (immediate) adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

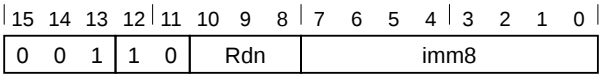
```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3>
  // Inside IT block
ADDS{<q>} <Rd>, <Rn>, #<imm3>
  // Outside IT block
```

### Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

### T2

*Armv8-M*




**T2 variant**

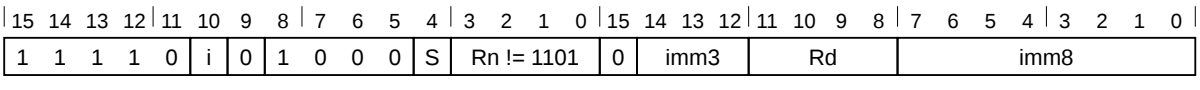
```
ADD<c>{<q>} <Rdn>, #<imm8>
  // Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8>
  // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8>
  // Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8>
  // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

### Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

### T3

*Armv8-M Main Extension only*




**ADD variant**

**Applies when `S == 0`.**

```
ADD<c>.W {<Rd>,} <Rn>, #<const>
  // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### ADDS variant

**Applies when `S == 1 && Rd != 1111`.**

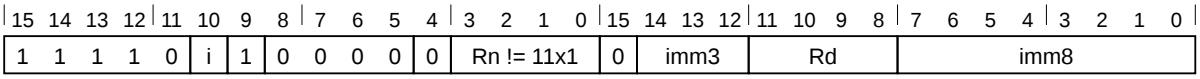
```
ADDS.W {<Rd>,} <Rn>, #<const>
  // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

## Decode for this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '1101' then SEE "ADD (SP plus immediate)"
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

## T4

*Armv8-M Main Extension only*




### T4 variant

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, #<imm12>
  // <imm12> cannot be represented in T1, T2, or T3
ADDW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12>
  // <imm12> can be represented in T1, T2, or T3
```

## Decode for this encoding

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE "ADD (SP plus immediate)"
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the general-purpose source and destination register, encoded in the "Rdn" field. |
| <imm8> | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | For encoding T1: is the general-purpose source register, encoded in the "Rn" field.<br>For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.3 *ADD (SP plus immediate)* on page 503.<br>For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.3 *ADD (SP plus immediate)* on page 503. If the PC is used, see C2.4.8 *ADR* on page 515. |
| <imm3> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field. |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

```
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;
```

## C2.4.6 ADD (immediate, to PC)

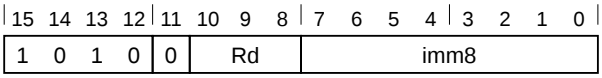
Add to PC. Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This instruction is a pseudo-instruction of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of ADR gives the operational pseudocode for this instruction.

### T1

*Armv8-M*




**T1 variant**

```
ADD{<c>}{<q>} <Rd>, PC, #<imm8>
```

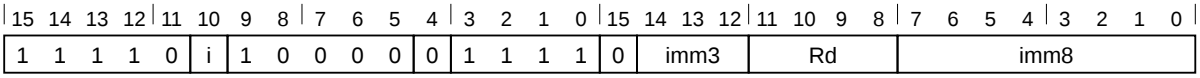
**is equivalent to**

```
ADR{<c>}{<q>} <Rd>, <label>
```

**and is never the preferred disassembly.**

### T3

*Armv8-M Main Extension only*




**T3 variant**

```
ADDW{<c>}{<q>} <Rd>, PC, #<imm12>
  // <Rd>, <imm12> can be represented in T1
```

**is equivalent to**

```
ADR{<c>}{<q>} <Rd>, <label>
```

**and is never the preferred disassembly.**

**T3 variant**

```
ADD{<c>}{<q>} <Rd>, PC, #<imm12>
```

**is equivalent to**

```
ADR{<c>}{<q>} <Rd>, <label>
```

**and is never the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |

| | |
|---|---|
| <label> | For encoding T1: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.<br>For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label. If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset. If the offset is negative, encoding T2 is used, with imm32 equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of imm32. Permitted values of the size of the offset are 0-4095. |
| <imm8> | Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4. |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |

## Operation for all encodings

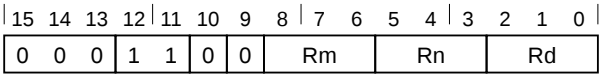
The description of ADR gives the operational pseudocode for this instruction.

## C2.4.7 ADD (register)

Add (register). ADD (register) adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




#### T1 variant

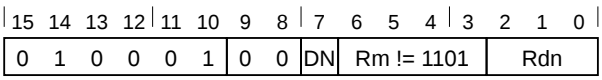
```
ADD<c>{<q>} <Rd>, <Rn>, <Rm>
  // Inside IT block
ADDS{<q>} {<Rd>,} <Rn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
2  (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M*




#### T2 variant

**Applies when !(DN == 1 && Rdn == 101).**

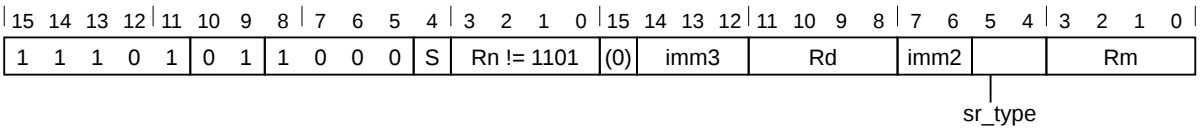
```
ADD<c>{<q>} <Rdn>, <Rm>
  // Preferred syntax, Inside IT block
ADD{<c>}{<q>} {<Rdn>,} <Rdn>, <Rm>
```

### Decode for this encoding

```
1  if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)"
2  d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
3  (shift_t, shift_n) = (SRType_LSL, 0);
4  if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
5  if d == 15 && m == 15 then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### ADD, rotate right with extend variant

**Applies when S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11.**

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ADD, shift or rotate by value variant**

**Applies when S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11).**

```
ADD<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**ADDS, rotate right with extend variant**

**Applies when S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11.**

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ADDS, shift or rotate by value variant**

**Applies when S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111.**

```
ADDS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1 or T2
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "CMN (register)";
2  if Rn == '1101' then SEE "ADD (SP plus register)"
3  if !HaveMainExt() then UNDEFINED;
4  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
5  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
6  if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rdn> Is the general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch. The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.

<Rd> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:
- If omitted, this register is the same as <Rn>.
- If present, encoding T1 is preferred to encoding T2.

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.4 *ADD (SP plus register)* on page 505.

<Rm> For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.
For encoding T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used.

<shift> Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:

LSL when sr_type = 00
LSR when sr_type = 01
ASR when sr_type = 10
ROR when sr_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

```
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
        BranchTo(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;
```

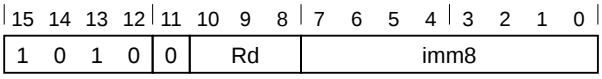
## C2.4.8 ADR

Form PC-relative address. Address to Register adds an immediate value to the PC value, and writes the result to the destination register.

This instruction is used by the pseudo-instructions ADD (immediate, to PC) and SUB (immediate, from PC).

### T1

*Armv8-M*




#### T1 variant

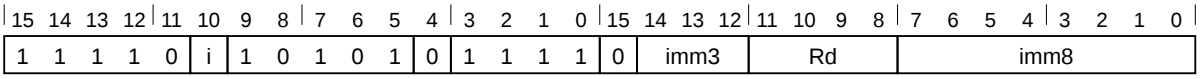
```
ADR{<c>}{<q>} <Rd>, <label>
```

### Decode for this encoding

```
1  d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

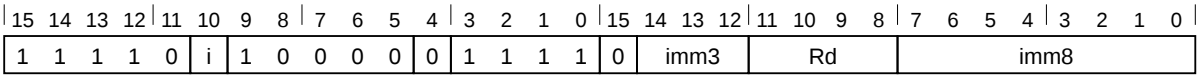
```
ADR{<c>}{<q>} <Rd>, <label>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
3  if d IN {13,15} then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

```
ADR{<c>}.W <Rd>, <label>
  // <Rd>, <label> can be presented in T1
ADR{<c>}{<q>} <Rd>, <label>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
3  if d IN {13,15} then UNPREDICTABLE;
```

### Alias conditions

| Alias or pseudo-instruction | preferred when |
|---|---|
| ADD (immediate, to PC) | Never |
| SUB (immediate, from PC) | `i:imm3:imm8 == '000000000000'` |

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<label>` | For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the Align `(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.<br>For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the Align `(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095. |

## Operation for all encodings

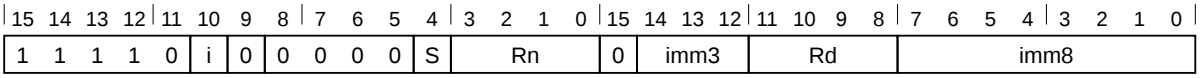
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
4     R[d] = result;
```

## C2.4.9 AND (immediate)

Bitwise AND (immediate). AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

### T1

*Armv8-M Main Extension only*




**AND variant**

**Applies when `S == 0`.**

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**ANDS variant**

**Applies when `S == 1 && Rd != 1111`.**

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "TST (immediate)";
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
4  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5  if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

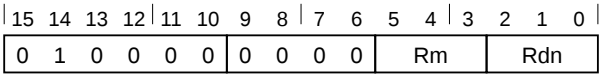
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = R[n] AND imm32;
4      R[d] = result;
5      if setflags then
6          APSR.N = result<31>;
7          APSR.Z = IsZeroBit(result);
8          APSR.C = carry;
9          // APSR.V unchanged
```

### C2.4.10 AND (register)

Bitwise AND (register). AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

#### T1

*Armv8-M*




**T1 variant**

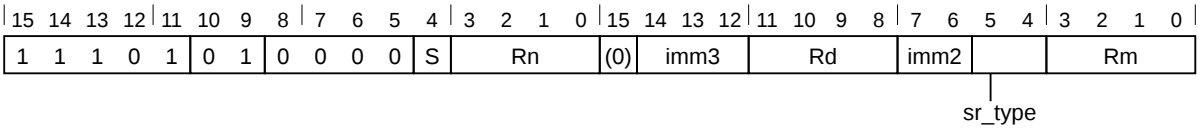
```
AND<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
ANDS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

#### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

#### T2

*Armv8-M Main Extension only*




**AND, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**AND, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
AND<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**ANDS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11`.**

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ANDS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111`.**

```
ANDS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if Rd == '1111' && S == '1' then SEE "TST (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

```
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
```

## C2.4.11 ASR (immediate)

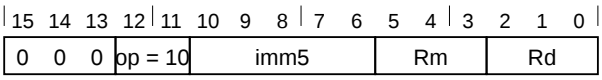
Arithmetic Shift Right (immediate). Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*




**T2 variant**

```
ASR<c>{<q>} {<Rd>,} <Rm>, #<imm>
  // Inside IT block
```

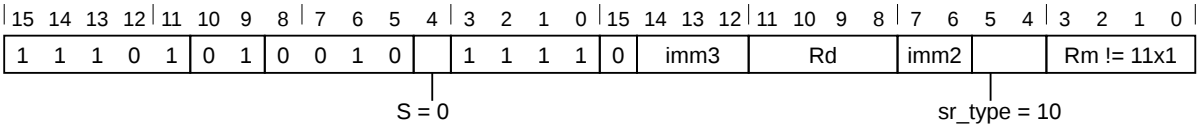
**is equivalent to**

```
MOV<c>{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOV, shift or rotate by value variant**

```
ASR<c>.W {<Rd>,} <Rm>, #<imm>
  // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

**MOV, shift or rotate by value variant**

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

| | |
|---|---|
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.12 ASR (register)

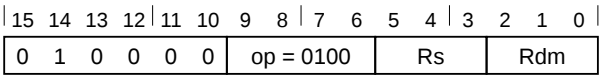
Arithmetic Shift Right (register). Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination registers. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*




**Arithmetic shift right variant**

```
ASR<c>{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Inside IT block
```

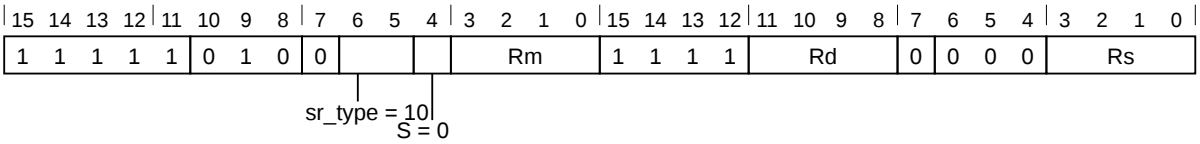
**is equivalent to**

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Non flag setting variant**

```
ASR<c>.W {<Rd>,} <Rm>, <Rs>
  // Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

**Non flag setting variant**

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

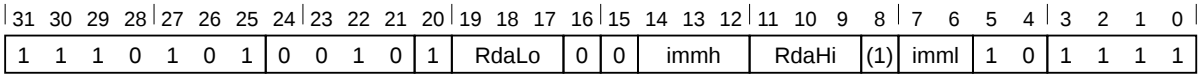
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.13 ASRL (immediate)

Arithmetic Shift Right Long. Arithmetic shift right by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: ASRL variant

```
ASRL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "SRSHR (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

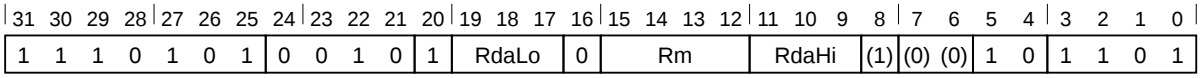
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = SInt(R[dah]:R[dal]);
5      result = (op1 >> amount)<63:0>;
6      R[dah] = result<63:32>;
7      R[dal] = result<31:0>;
```

## C2.4.14 ASRL (register)

Arithmetic Shift Right Long. Arithmetic shift right by 0 to 64 bits of a 64 bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: ASRL variant

```
ASRL<c> RdaLo, RdaHi, Rm
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "SQRSHR (register)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  m   = UInt(Rm);
8  if RdaHi == '110' || Rm == '11x1' || Rm == RdaHi:'1' then CONSTRAINED_UNPREDICTABLE;
9  if Rm    == RdaLo:'0' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      amount = SInt(R[m]<7:0>);
5      op1    = SInt(R[dah]:R[dal]);
6      result = (op1 >> amount)<63:0>;
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

## C2.4.15 ASRS (immediate)

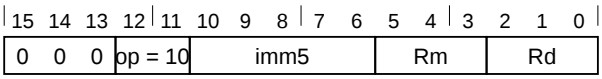
Arithmetic Shift Right, Setting flags (immediate). Arithmetic Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M*




**T2 variant**

```
ASRS{<q>} {<Rd>,} <Rm>, #<imm>
  // Outside IT block
```

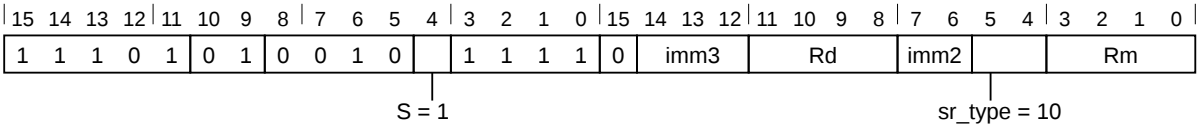
**is equivalent to**

```
MOVS{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOVS, shift or rotate by value variant**

```
ASRS.W {<Rd>,} <Rm>, #<imm>
  // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

**MOVS, shift or rotate by value variant**

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.16 ASRS (register)

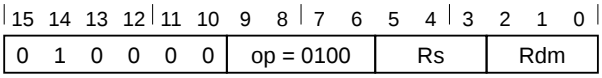
Arithmetic Shift Right, Setting flags (register). Arithmetic Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M*




**Arithmetic shift right variant**

```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Outside IT block
```

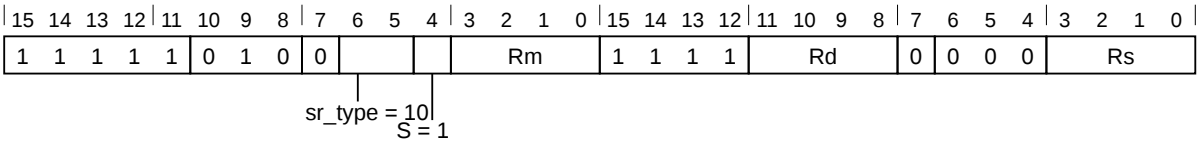
**is equivalent to**

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Flag setting variant**

```
ASRS.W {<Rd>,} <Rm>, <Rs>
  // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

**Flag setting variant**

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

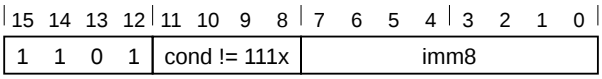
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.17 B

Branch. Branch causes a branch to a target address.

### T1

*Armv8-M*




#### T1 variant

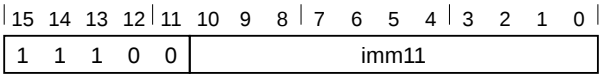
```
B<c>{<q>} <label>
  // Not permitted in IT block
```

### Decode for this encoding

```
1 if cond == '1110' then SEE UDF;
2 if cond == '1111' then SEE SVC;
3 imm32 = SignExtend(imm8:'0', 32);
4 if InITBlock() then UNPREDICTABLE;
```

### T2

*Armv8-M*




#### T2 variant

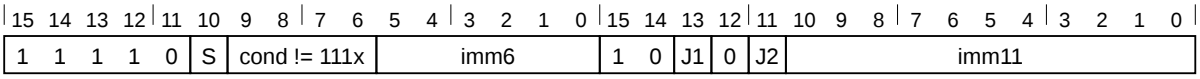
```
B{<c>}{<q>} <label>
  // Outside or last in IT block
```

### Decode for this encoding

```
1 imm32 = SignExtend(imm11:'0', 32);
2 cond = CurrentCond();
3 if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

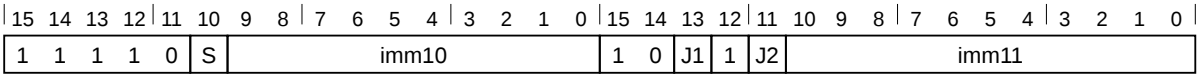
```
B<c>.W <label>
  // Not permitted in IT block, and <label> can be represented in T1
B<c>{<q>} <label>
  // Not permitted in IT block
```

### Decode for this encoding

```
if cond<3:1> == '111' then SEE "Related encodings";
if !HaveMainExt() then UNDEFINED;
imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

### T4

*Armv8-M*




#### T4 variant

```
B{<c>}.W <label>
  // <label> can be represented in T2
B{<c>}{<q>} <label>
```

### Decode for this encoding

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
cond = CurrentCond();
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | For encoding T1: see C1.2.5 *Standard assembler syntax fields* on page 424. Must not be AL or omitted.<br>For encoding T2 and T4: see C1.2.5 *Standard assembler syntax fields* on page 424.<br>For encoding T3: see C1.2.5 *Standard assembler syntax fields* on page 424. <c> must not be AL or omitted. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <label> | For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range -256 to 254.<br>For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range -2048 to 2046.<br>For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range -1048576 to 1048574.<br>For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range -16777216 to 16777214. |

### Operation for all encodings

```
if ConditionPassed(cond) then
    EncodingSpecificOperations();
    BranchTo(PC + imm32);
```

## C2.4.18 BF, BFX, BFL, BFLX, BFCSEL

Branch Future, Branch Future and Exchange, Branch Future with Link, Branch Future with Link and Exchange, Branch Future Conditional Select. These instructions notify the PE about an upcoming branch to `<label>`, so that the branch will be taken instead of fetching and executing the instruction at `<b_label>`. This allows the PE to reduce or eliminate any associated performance penalty that might have been caused by a branch that the PE was not notified about. It is IMPLEMENTATION DEFINED whether this instruction is treated as a NOP. To ensure correct program flow behavior in these cases, fallback code should be present at the point specified by `<b_label>`.

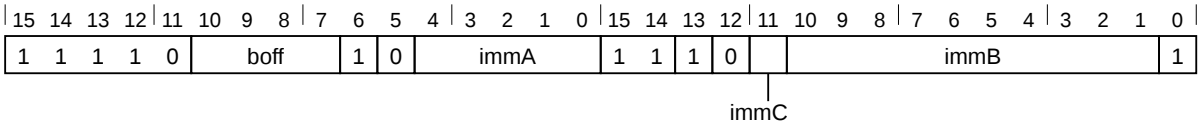
For the Branch Future with Link variant, the link register is updated when the branch is performed. The value written is offset from the branch point by 4 bytes, which corresponds to the length of the BL (immediate) instruction that would usually follow the branch point as part of the fallback code.

For the Branch Future with Link and Exchange variant, the link register is updated when the branch is performed. The value written is offset from the branch point by 2 bytes, which corresponds to the length of the BLX (register) instruction that would usually follow the branch point as part of the fallback code.

The Branch Future Conditional Select variant creates a future branch to `<label>` if the condition code passes. If the explicit condition code fails, this variant does not behave as a NOP. Instead it creates a future branch to the instruction specified by `<ba_label>`, if there is no other active BF entry in the loop and branch cache.

### T1

*Armv8.1-M Low Overhead Branch Extension*




#### T1: BF variant

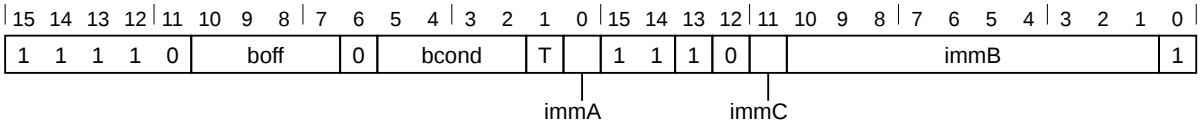
```
BF<c> <b_label>, <label>
```

### Decode for this encoding

```
1  if boff == '0000' then SEE "Related encodings";
2  if !HaveLOBExt()  then UNDEFINED;
3
4  endOffset = ZeroExtend(boff:'0', 32);
5  offset    = SignExtend(immA:immB:immC:'1', 32);
6  bOffset   = bits(32) UNKNOWN;
7  n         = integer UNKNOWN;
8  regAddr   = FALSE;
9  link      = FALSE;
10 bcond     = CondAL;
```

### T2

*Armv8.1-M Low Overhead Branch Extension*




#### T2: BFCSEL variant

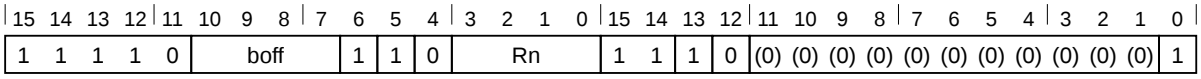
```
BFCSEL <b_label>, <label>, <ba_label>, <bcond>
```

### Decode for this encoding

```
if boff == '0000' then SEE "Related encodings";
if !HaveLOBExt()  then UNDEFINED;

endOffset = ZeroExtend(boff:'0', 32);
offset    = SignExtend(immA:immB:immC:'1', 32);
bOffset   = (if T == '1' then 5 else 3)<31:0>;
n         = integer UNKNOWN;
regAddr   = FALSE;
link      = FALSE;
if InITBlock()     then CONSTRAINED_UNPREDICTABLE;
if bcond == '111x' then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M Low Overhead Branch Extension*




### T3: BFX variant

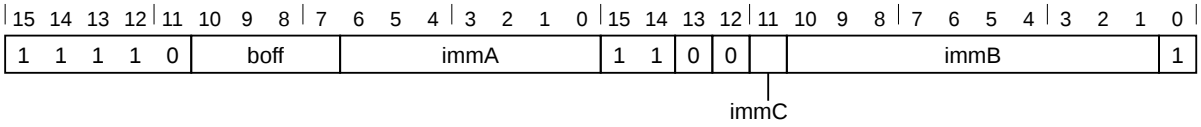
```
BFX<c> <b_label>, Rn
```

### Decode for this encoding

```
if boff == '0000' then SEE "Related encodings";
if !HaveLOBExt()  then UNDEFINED;

endOffset = ZeroExtend(boff:'0', 32);
offset    = Zeros(32);
bOffset   = bits(32) UNKNOWN;
n         = UInt(Rn);
regAddr   = TRUE;
link      = FALSE;
bcond     = CondAL;
if Rn == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M Low Overhead Branch Extension*




### T4: BFL variant

```
BFL<c> <b_label>, <label>
```

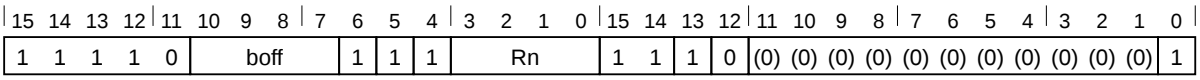
### Decode for this encoding

```
if boff == '0000' then SEE "Related encodings";
if !HaveLOBExt()  then UNDEFINED;

endOffset = ZeroExtend(boff:'0', 32);
offset    = SignExtend(immA:immB:immC:'1', 32);
bOffset   = bits(32) UNKNOWN;
n         = integer UNKNOWN;
regAddr   = FALSE;
```

```
9  link      = TRUE;
10 bcond     = CondAL;
```

## T5

*Armv8.1-M Low Overhead Branch Extension*




### T5: BFLX variant

```
BFLX<c> <b_label>, Rn
```

## Decode for this encoding

```
1  if boff == '0000' then SEE "Related encodings";
2  if !HaveLOBExt()  then UNDEFINED;
3
4  endOffset = ZeroExtend(boff:'0', 32);
5  offset    = Zeros(32);
6  bOffset   = bits(32) UNKNOWN;
7  n         = UInt(Rn);
8  regAddr   = TRUE;
9  link      = TRUE;
10 bcond     = CondAL;
11 if Rn == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <b_label> | The PC relative offset of the first instruction in the fallback code, that will not be executed if the future branch is taken. |
| <Rn> | The address to branch to. |
| <ba_label> | The PC relative offset of the address to branch to in case the associated BFCSEL condition code check fails and no other branch future is pending. The range of this address allows branching over a 2-byte or 4-byte instruction located at <b_label>. |
| <T> | Selects whether the instruction at <b_label> is a 2-byte (T = 0) or 4-byte (T = 1) instruction to be branched around, as specified by <ba_label>. |
| <bcond> | The comparison condition to use. The evaluation of this comparison is performed when this instruction is executed and not at the point the branch is performed.<br>This parameter must be one of the following values:<br>EQ Encoded as bcond = 0000<br>NE Encoded as bcond = 0001<br>CS Encoded as bcond = 0010<br>CC Encoded as bcond = 0011<br>MI Encoded as bcond = 0100<br>PL Encoded as bcond = 0101<br>VS Encoded as bcond = 0110<br>VC Encoded as bcond = 0111<br>HI Encoded as bcond = 1000<br>LS Encoded as bcond = 1001<br>GE Encoded as bcond = 1010<br>LT Encoded as bcond = 1011<br>GT Encoded as bcond = 1100<br>LE Encoded as bcond = 1101 |
| <label> | The PC relative offset of the address to branch to. |

## Operation for all encodings

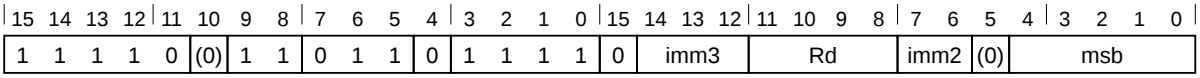
```
if ConditionPassed() then
    EncodingSpecificOperations();

    // BF can be implemented as a NOP
    allow_bf = boolean IMPLEMENTATION_DEFINED "BF allowed";

    if ConditionPassed(bcond) then
        jump_addr = if regAddr then R[n] else PC + offset;
    else
        jump_addr = PC + endOffset + bOffset;
        // If the condition fails, capture a BF entry only if it does not override
        // an existing BF entry.
        if LO_BRANCH_INFO.VALID == '1' && LO_BRANCH_INFO.BF == '1' then
            allow_bf = FALSE;
    // A branch that results in a transition to a different instruction set
    // causes the BF instruction to be treated as a NOP.
    allow_bf = allow_bf && (jump_addr<0> == '1');
    // Branches that would cause a FNC_RETURN or EXC_RETURN aren't supported,
    // and also cause BF to be treated as a NOP.
    allow_bf = allow_bf && (!regAddr || IsReturn(jump_addr) == AddrType_NORMAL);
    // Check if the branch cache info is enabled.
    allow_bf = allow_bf && CCR.LOB == '1';
    // Set up the branch info cache if allowed
    if allow_bf then
        LO_BRANCH_INFO.VALID     = '1';
        LO_BRANCH_INFO.BF        = '1';
        LO_BRANCH_INFO.LF        = if link    then '1' else '0';
        LO_BRANCH_INFO.T16IND    = if regAddr then '1' else '0';
        LO_BRANCH_INFO.END_ADDR  = (PC + endOffset)<31:1>;
        LO_BRANCH_INFO.JUMP_ADDR = jump_addr<31:1>;
```

## C2.4.19 BFC

Bit Field Clear. Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
3  if msbit < lsbit then UNPREDICTABLE;
4  if d IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <lsb> | Is the least significant bit that is to be cleared, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the number of bits to be cleared, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |

### Operation for all encodings

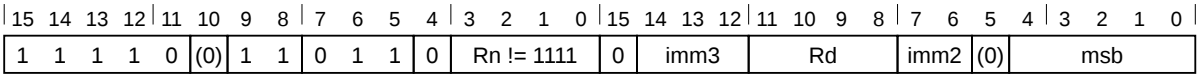
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if msbit >= lsbit then
4          R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
5          // Other bits of R[d] are unchanged
6      else
7          R[d] = bits(32) UNKNOWN;
```

## C2.4.20 BFI

Bit Field Insert. Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE BFC;
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
4  if msbit < lsbit then UNPREDICTABLE;
5  if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `msbit < lsbit`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | Is the least significant destination bit, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the number of bits to be copied, in the range 1 to 32-<lsb>, encoded in the "msb" field as <lsb>+<width>-1. |

### Operation for all encodings

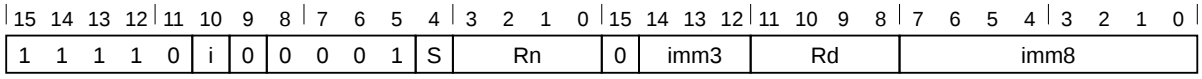
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if msbit >= lsbit then
4          R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
5          // Other bits of R[d] are unchanged
6      else
7          R[d] = bits(32) UNKNOWN;
```

## C2.4.21 BIC (immediate)

Bit Clear (immediate). Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**BIC variant**

**Applies when `S == 0`.**

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**BICS variant**

**Applies when `S == 1`.**

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
3  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

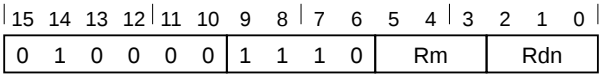
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = R[n] AND NOT(imm32);
4      R[d] = result;
5      if setflags then
6          APSR.N = result<31>;
7          APSR.Z = IsZeroBit(result);
8          APSR.C = carry;
9          // APSR.V unchanged
```

## C2.4.22 BIC (register)

Bit Clear (register). Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

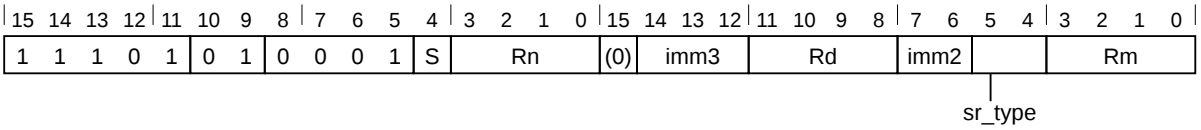
```
BIC<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
BICS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




**BIC, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**BIC, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
BIC<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**BICS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**BICS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
BICS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

```
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
```

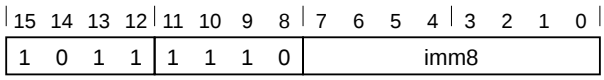
## C2.4.23 BKPT

Breakpoint. Breakpoint causes a DebugMonitor exception or a debug halt to occur depending on the configuration of the debug support.

BKPT is an unconditional instruction and executes as such both inside and outside an IT instruction block.

### T1

*Armv8-M*




#### T1 variant

```
BKPT{<q>} {#}<imm>
```

### Decode for this encoding

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly only and is ignored by hardware.
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. A BKPT instruction must be unconditional. |
| <imm> | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores this value, but a debugger might use it to store additional information about the breakpoint. |

### Operation for all encodings

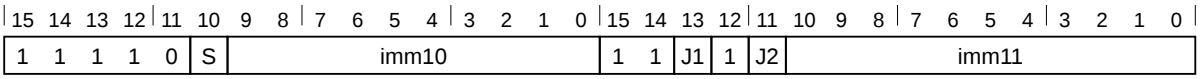
```
EncodingSpecificOperations();
GenerateDebugEventResponse(TRUE);
```

## C2.4.24 BL

Branch with Link (immediate). Branch with Link (immediate) calls a subroutine at a PC-relative address.

### T1

*Armv8-M*




#### T1 variant

```
BL{<c>}{<q>} <label>
```

### Decode for this encoding

```
1  I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
2  if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <label> | The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding with imm32 set to that offset. Permitted offsets are even numbers in the range -16777216 to 16777214. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      next_instr_addr = PC;
4      LR = next_instr_addr<31:1> : '1';
5      BranchTo(PC + imm32);
```

## C2.4.25 BLX, BLXNS

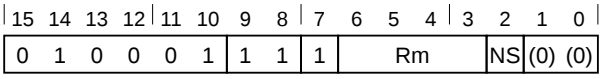
Branch with Link and Exchange (Non-secure). Branch with Link and Exchange calls a subroutine at an address, with the address and instruction set specified by a register. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

Branch with Link and Exchange Non-secure calls a subroutine at an address specified by a register, and if bit[0] of the target address is 0 then the instruction causes a transition from Secure to Non-secure state. This variant of the instruction must only be used when the additional steps required to make such a transition safe have been taken.

BLXNS is UNDEFINED if executed in Non-secure state, or if the Security Extension is not implemented.

### T1

*Armv8-M*




**BLX variant**

**Applies when `NS == 0`.**

```
BLX{<c>}{<q>} <Rm>
```

**BLXNS variant**

**Applies when `NS == 1`.**

```
BLXNS{<c>}{<q>} <Rm>
```

### Decode for this encoding

```
m = UInt(Rm); allowNonSecure = NS == '1';
if !IsSecure() && allowNonSecure then UNDEFINED;
if m IN {13,15} then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rm> | Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    target        = R[m];
    nextInstrAddr = NextInstrAddr()<31:1> : '1';

    if allowNonSecure && (target<0> == '0') then
        if !IsAligned(SP, 8) then UNPREDICTABLE;
        address = SP - 8;
        RETPSR_Type savedPSR = Zeros();
        savedPSR.Exception   = IPSR.Exception;
        savedPSR.SFPA        = CONTROL_S.SFPA;
        // Only the stack locations, not the store order, are architected
```

```
        spName = LookUpSP();
        mode   = CurrentMode();
        exc                              = Stack(address, 0, spName, mode, nextInstrAddr);
        if exc.fault == NoFault then exc = Stack(address, 4, spName, mode, savedPSR);
        HandleException(exc);
        // Stack pointer update will raise a fault if limit violated
        SP = address;
        LR = 0xFEFFFFFF<31:0>;
        // If in handler mode, IPSR must be non-zero. To prevent revealing which
        // Secure handler is calling Non-secure code, IPSR is set to an invalid but
        // non-zero value(ie the reset exception number).
        if mode == PEMode_Handler then
            IPSR = 0x1<31:0>;
    else
        LR = nextInstrAddr;

    BranchCall(target, allowNonSecure);
```

### CONSTRAINED UNPREDICTABLE behavior

If `!IsAligned(SP, 8)`, then one of the following behaviors must occur:

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

## C2.4.26 BX, BXNS

Branch and Exchange (Non-secure). Branch and Exchange causes a branch to an address, with the address and instruction set specified by a register. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

Branch and Exchange Non-secure causes a branch to an address specified by a register. If bit[0] of the target address is 0, and the target address is not FNC_RETURN or EXC_RETURN, then the instruction causes a transition from Secure to Non-secure state. This variant of the instruction must only be used when the additional steps required to make such a transition safe have been taken.

BX can also be used for an exception return.

BXNS is UNDEFINED if executed in Non-secure state, or if the Security Extension is not implemented.

### T1

*Armv8-M*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | Rm | | | | NS | (0) | (0) |

**BX variant**

**Applies when `NS == 0`.**

```
BX{<c>}{<q>} <Rm>
```

**BXNS variant**

**Applies when `NS == 1`.**

```
BXNS{<c>}{<q>} <Rm>
```

### Decode for this encoding

```
1  m = UInt(Rm); allowNonSecure = NS == '1';
2  if !IsSecure() && allowNonSecure then UNDEFINED;
3  if m IN {13,15} then UNPREDICTABLE;
4  if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rm> | Is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The SP can be used, but this is deprecated. |

### Operation for all encodings

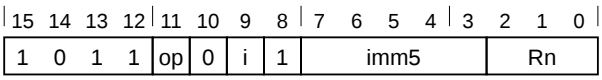
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      exc = BranchReturn(R[m], allowNonSecure);
4      HandleException(exc);
```

## C2.4.27 CBNZ, CBZ

Compare and Branch on Nonzero or Zero. Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

### T1

*Armv8-M*




**CBNZ variant**

**Applies when `op == 1`.**

```
CBNZ{<q>} <Rn>, <label>
```

**CBZ variant**

**Applies when `op == 0`.**

```
CBZ{<q>} <Rn>, <label>
```

### Decode for this encoding

```
1 n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
2 if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose register to be tested, encoded in the "Rn" field. |
| <label> | Is the program label to be conditionally branched to. Its offset from the PC, a multiple of 2 in the range 0 to 126, is encoded as "i:imm5" times 4. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 if nonzero != IsZero(R[n]) then
3     BranchTo(PC + imm32);
```

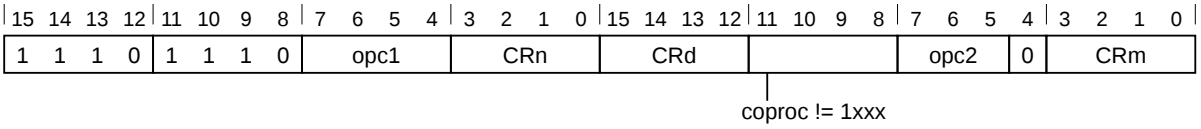
## C2.4.28 CDP, CDP2

Coprocessor Data Processing. Coprocessor Data Processing tells a coprocessor to perform an operation.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

### T1

*Armv8-M Main Extension only*




#### T1 variant

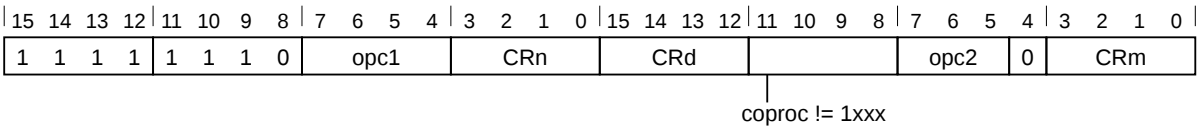
```
CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  cp = UInt(coproc);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  cp = UInt(coproc);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p0 to p7, p10, and p11. |
| <opc1> | Is a coprocessor-specific opcode, in the range 0 to 15, encoded in the "opc1" field. |
| <CRd> | Is the destination coprocessor register, encoded in the "CRd" field. |
| <CRn> | Is the coprocessor register that contains the first operand, encoded in the "CRn" field. |
| <CRm> | Is the coprocessor register that contains the second operand, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_InternalOperation(cp, ThisInstr());
```

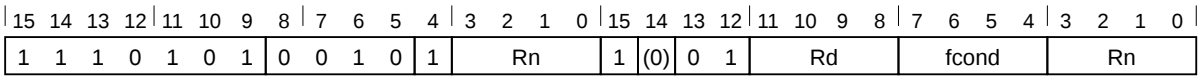
## C2.4.29 CINC

Conditional Increment. Returns, in the destination register, the value of the source register incremented by 1, if the condition is TRUE. Otherwise returns the value of the source register.

This is an alias of CSINC with the following condition satisfied: Rn == Rm && Rn != 15.

This alias is the preferred disassembly.

### T1




**CINC variant**

```
CINC Rd, Rn, <fcond>
```

**is equivalent to**

```
CSINC Rd, Rn, Rn, invert (<cond>)
```

**and is the preferred disassembly when Rn == Rm && Rn != 15**

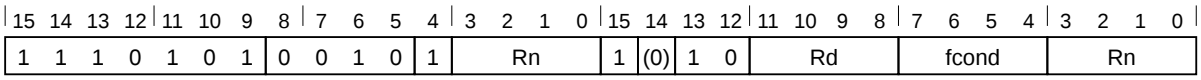
## C2.4.30 CINV

Conditional Invert. Returns, in the destination register, the bitwise inversion of the value of the source register, if the condition is TRUE. Otherwise returns the value of the source register.

This is an alias of CSINV with the following condition satisfied: Rn == Rm && Rn != 15.

This alias is the preferred disassembly.

### T1




**T1: CINV variant**

```
CINV Rd, Rn, <fcond>
```

**is equivalent to**

```
CSINV Rd, Rn, Rn, invert (<cond>)
```

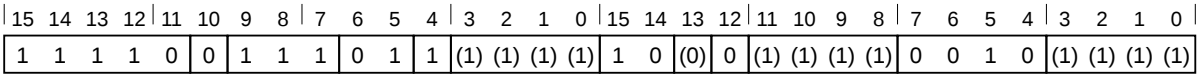
**and is the preferred disassembly when Rn == Rm && Rn != 15**

## C2.4.31 CLREX

Clear Exclusive. Clear Exclusive clears the local record of the executing PE that an address has had a request for an exclusive access.

### T1

*Armv8-M*




#### T1 variant

```
CLREX{<c>}{<q>}
```

### Decode for this encoding

```
// No additional decoding required
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

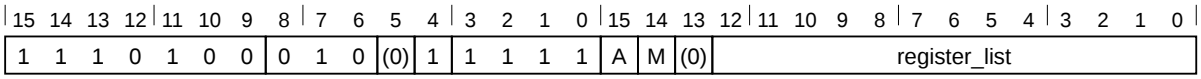
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

## C2.4.32 CLRM

Clear multiple. Zeros the specified general-purpose registers. It is IMPLEMENTATION DEFINED whether this instruction is exception-continuable. See EPSR.ICI. If an exception returns to this instruction with non-zero EPSR.ICI bits, and the PE does not support exception-continuable behavior, the instruction restarts from the beginning

### T1

*Armv8.1-M Main Extension and Security Extension*




#### T1: CLRM variant

```
CLRM<c> <registers>
```

### Decode for this encoding

```
1  if !HaveMainExt()           then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3
4  registers = A:M:'0':register_list;
5  if BitCount(registers) < 1 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <registers> | A list of the registers to clear. The valid registers are APSR, LR/R14, and R12-R0, and are encoded as a bitmask in the A, M and register_list fields. |

### Operation for all encodings

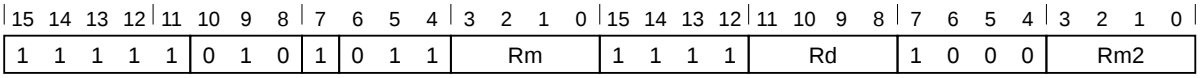
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      for i = 0 to 14
5          if registers<i> == '1' then
6              R[i] = Zeros(32);
7
8      if registers<15> == '1' then
9          APSR = Zeros(32);
```

## C2.4.33 CLZ

Count Leading Zeros. Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`CLZ{<c>}{<q>} <Rd>, <Rm>`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if Rm != Rm2 then UNPREDICTABLE;
3  d = UInt(Rd);  m = UInt(Rm);
4  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `Rm != Rm2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

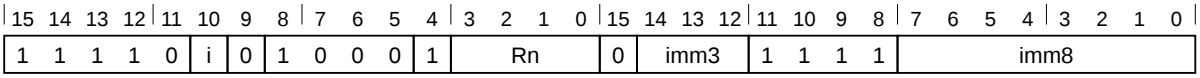
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = CountLeadingZeroBits(R[m]);
4      R[d] = result<31:0>;
```

## C2.4.34 CMN (immediate)

Compare Negative (immediate). Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
3  if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

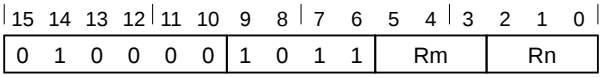
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
4      APSR.N = result<31>;
5      APSR.Z = IsZeroBit(result);
6      APSR.C = carry;
7      APSR.V = overflow;
```

## C2.4.35 CMN (register)

Compare Negative (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.

### T1

*Armv8-M*




#### T1 variant

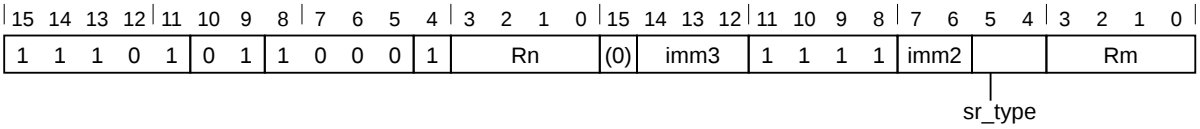
```
CMN{<c>}{<q>} <Rn>, <Rm>
```

### Decode for this encoding

```
1  n = UInt(Rn);  m = UInt(Rm);
2  (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### Rotate right with extend variant

**Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### Shift or rotate by value variant

**Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
CMN{<c>}.W <Rn>, <Rm>
  // <Rn>, <Rm> can be represented in T1
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  m = UInt(Rm);
3  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4  if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values: |

LSL when sr_type = 00
LSR when sr_type = 01
ASR when sr_type = 10
ROR when sr_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

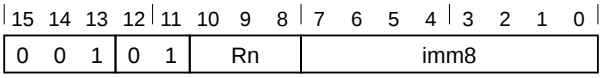
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

## C2.4.36 CMP (immediate)

Compare (immediate). Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M*




#### T1 variant

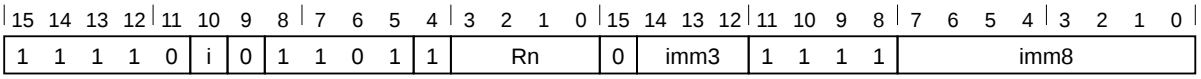
```
CMP{<c>}{<q>} <Rn>, #<imm8>
```

### Decode for this encoding

```
1  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
CMP{<c>}.W <Rn>, #<const>
  // <Rn>, <const> can be represented in T1
CMP{<c>}{<q>} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
3  if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | For encoding T1: is a general-purpose source register, encoded in the "Rn" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rn" field. |
| <imm8> | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

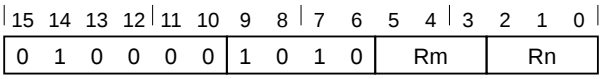
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
4      APSR.N = result<31>;
5      APSR.Z = IsZeroBit(result);
6      APSR.C = carry;
7      APSR.V = overflow;
```

## C2.4.37 CMP (register)

Compare (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.

### T1

*Armv8-M*




#### T1 variant

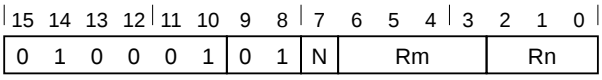
```
CMP{<c>}{<q>} <Rn>, <Rm>
  // <Rn> and <Rm> both from R0-R7
```

### Decode for this encoding

```
1 n = UInt(Rn);  m = UInt(Rm);
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M*




#### T2 variant

```
CMP{<c>}{<q>} <Rn>, <Rm>
  // <Rn> and <Rm> not both from R0-R7
```

### Decode for this encoding

```
1 n = UInt(N:Rn);  m = UInt(Rm);
2 (shift_t, shift_n) = (SRType_LSL, 0);
3 if n < 8 && m < 8 then UNPREDICTABLE;
4 if n == 15 || m == 15 then UNPREDICTABLE;
```

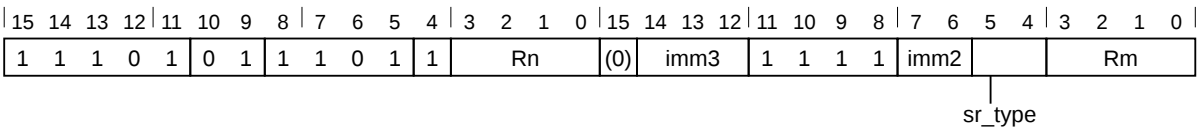
#### CONSTRAINED UNPREDICTABLE behavior

If `n < 8 && m < 8`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The condition flags become UNKNOWN.

### T3

*Armv8-M Main Extension only*



**Rotate right with extend variant**

**Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

**Shift or rotate by value variant**

**Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
CMP{<c>}.W <Rn>, <Rm>
  // <Rn>, <Rm> can be represented in T1 or T2
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
```

## Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  m = UInt(Rm);
3  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4  if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.<br>For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      shifted = Shift(R[m], shift_t, shift_n, APSR.C);
4      (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
5      APSR.N = result<31>;
6      APSR.Z = IsZeroBit(result);
7      APSR.C = carry;
8      APSR.V = overflow;
```

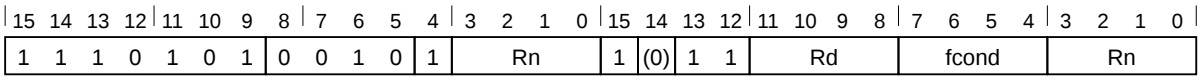
## C2.4.38 CNEG

Conditional Negate. Returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG with the following condition satisfied: Rn == Rm.

This alias is the preferred disassembly.

### T1




**CNEG variant**

```
CNEG Rd, Rn, <fcond>
```

**is equivalent to**

```
CSNEG Rd, Rn, Rn, invert (<cond>)
```

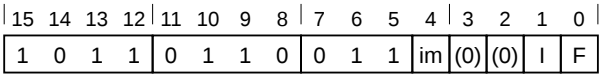
**and is the preferred disassembly when Rn == Rm**

## C2.4.39 CPS

Change PE State. Change PE State. The instruction modifies the PRIMASK and FAULTMASK special-purpose register values.

### T1

*Armv8-M*




**CPSID variant**

**Applies when `im == 1`.**

```
CPSID{<q>} <iflags>
```

**CPSIE variant**

**Applies when `im == 0`.**

```
CPSIE{<q>} <iflags>
```

### Decode for this encoding

```
enable = (im == '0');  disable = (im == '1');
if InITBlock() then UNPREDICTABLE;
if (I == '0' && F =='0') then UNPREDICTABLE;
affectPRI = (I == '1');  affectFAULT = (F == '1');
if !HaveMainExt() then
    if (I == '0') then UNPREDICTABLE;
    if (F == '1') then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `I == '0' && F == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

#### CONSTRAINED UNPREDICTABLE behavior

If `!HaveMainExt() && (I == '0' || F == '1')`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

### Assembler symbols for all encodings

`<q>` See C1.2.5 *Standard assembler syntax fields* on page 424.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

`f` FAULTMASK. When set to 1, raises the execution priority to -1, the same priority as HardFault. This is a 1-bit register, that can be updated only by privileged software. The register clears to 0 on return from any exception other than NMI.

`i` PRIMASK. When set to 1, raises the execution priority to 0. This is a 1-bit register, that can be updated only by privileged software.

## Operation for all encodings

```
EncodingSpecificOperations();
if CurrentModeIsPrivileged() then
    if enable then
        if affectPRI then
            PRIMASK.PM = '0';
        if affectFAULT then
            FAULTMASK.FM = '0';
    if disable then
        if affectPRI then
            PRIMASK.PM = '1';
        if affectFAULT && ExecutionPriority() > -1 then
            FAULTMASK.FM = '1';
```

## C2.4.40 CSDB

Consumption of Speculative Data Barrier. Consumption of Speculative Data Barrier is a memory barrier that controls speculative execution and data value prediction.

No instruction other than branch instructions and instructions that write to the PC appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- APSR.{N,Z,C,V} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

APSR.{N,Z,C,V} is not considered a data value. This instruction permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or APSR.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

### T1

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

#### T1 variant

`CSDB{<c>}.W`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if InITBlock() then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes unconditionally.
- The instruction executes conditionally.

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ConsumptionOfSpeculativeDataBarrier();
```

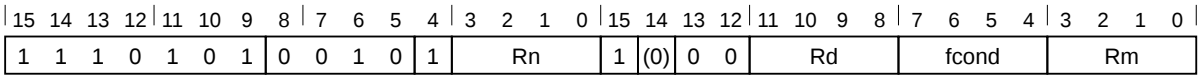
## C2.4.41 CSEL

Conditional Select. Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Main Extension only*




#### T1: CSEL variant

```
CSEL Rd, Rn, Rm, <fcond>
```

### Decode for this encoding

```
1  if Rm == '1101'              then SEE "Related encodings";
2  if !HaveMainExt()            then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  d = UInt(Rd);
5  m = UInt(Rm);
6  n = UInt(Rn);
7  if InITBlock()                    then CONSTRAINED_UNPREDICTABLE;
8  if Rd    == '11x1' || Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
9  if fcond == '111x'                 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <Rd> | Destination general-purpose register. |
| <Rn> | First source general-purpose register (ZR is permitted, PC is not). |
| <Rm> | Second source general-purpose register (ZR is permitted, PC is not). |
| <fcond> | The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values: |

| | | |
|---|---|---|
| EQ | Encoded as | fcond = 0000 |
| NE | Encoded as | fcond = 0001 |
| CS | Encoded as | fcond = 0010 |
| CC | Encoded as | fcond = 0011 |
| MI | Encoded as | fcond = 0100 |
| PL | Encoded as | fcond = 0101 |
| VS | Encoded as | fcond = 0110 |
| VC | Encoded as | fcond = 0111 |
| HI | Encoded as | fcond = 1000 |
| LS | Encoded as | fcond = 1001 |
| GE | Encoded as | fcond = 1010 |
| LT | Encoded as | fcond = 1011 |
| GT | Encoded as | fcond = 1100 |
| LE | Encoded as | fcond = 1101 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2
3  bits(32) result;
```

```
4  if ConditionHolds(fcond) then
5      result = RZ[n];
6  else
7      result = RZ[m];
8  R[d] = result;
```

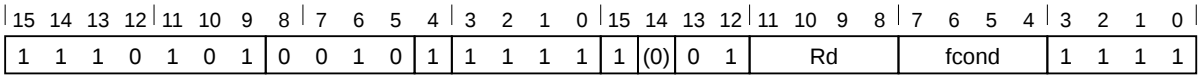
## C2.4.42 CSET

Conditional Set. Sets the destination register to 1 if the condition is TRUE, and otherwise set it to 0.

This is an alias of CSINC with the following condition satisfied: Rn==15 && Rm==15.

This alias is the preferred disassembly.

### T1




**CSET variant**

`CSET Rd, <fcond>`

**is equivalent to**

`CSINC Rd, Zr, Zr, invert (<cond>)`

**and is the preferred disassembly when Rn == 15 && Rm == 15**

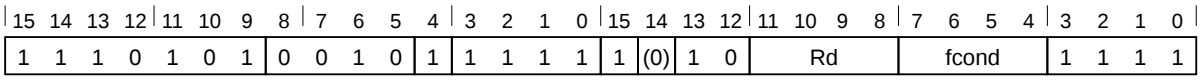
## C2.4.43 CSETM

Conditional Set Mask. Sets all bits of the destination register to 1 if the condition is TRUE. Otherwise sets all bits to 0.

This is an alias of CSINV with the following condition satisfied: Rn==15 && Rm==15.

This alias is the preferred disassembly.

### T1




**T1: CSETM variant**

```
CSETM Rd, <fcond>
```

**is equivalent to**

```
CSINV Rd, Zr, Zr, invert (<cond>)
```

**and is the preferred disassembly when Rn == 15 && Rm == 15**

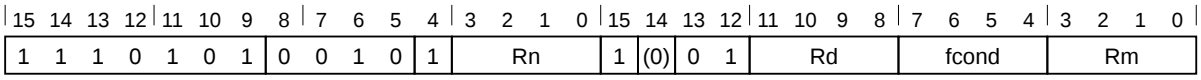
## C2.4.44 CSINC

Conditional Select Increment. Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Main Extension only*




#### T1: CSINC variant

```
CSINC Rd, Rn, Rm, <fcond>
```

### Decode for this encoding

```
1  if Rm == '1101'              then SEE "Related encodings";
2  if !HaveMainExt()            then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  d = UInt(Rd);
5  m = UInt(Rm);
6  n = UInt(Rn);
7  if InITBlock()                    then CONSTRAINED_UNPREDICTABLE;
8  if Rd    == '11x1' || Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
9  if fcond == '111x'                 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <Rd> | Destination general-purpose register. |
| <Rn> | First source general-purpose register (ZR is permitted, PC is not). |
| <Rm> | Second source general-purpose register (ZR is permitted, PC is not). |
| <fcond> | The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values: |

| | | |
|---|---|---|
| EQ | Encoded as | fcond = 0000 |
| NE | Encoded as | fcond = 0001 |
| CS | Encoded as | fcond = 0010 |
| CC | Encoded as | fcond = 0011 |
| MI | Encoded as | fcond = 0100 |
| PL | Encoded as | fcond = 0101 |
| VS | Encoded as | fcond = 0110 |
| VC | Encoded as | fcond = 0111 |
| HI | Encoded as | fcond = 1000 |
| LS | Encoded as | fcond = 1001 |
| GE | Encoded as | fcond = 1010 |
| LT | Encoded as | fcond = 1011 |
| GT | Encoded as | fcond = 1100 |
| LE | Encoded as | fcond = 1101 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2
3  bits(32) result;
```

```
4  if ConditionHolds(fcond) then
5      result = RZ[n];
6  else
7      result = RZ[m] + 1;
8  R[d] = result;
```

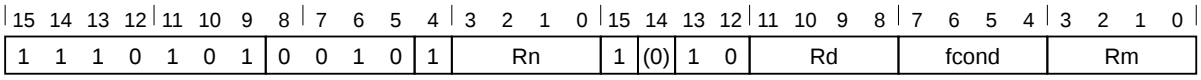
## C2.4.45 CSINV

Conditional Select Invert. Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register, bitwise inverted.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Main Extension only*




#### T1: CSINV variant

```
CSINV Rd, Rn, Rm, <fcond>
```

### Decode for this encoding

```
1  if Rm == '1101'              then SEE "Related encodings";
2  if !HaveMainExt()            then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  d = UInt(Rd);
5  m = UInt(Rm);
6  n = UInt(Rn);
7  if InITBlock()                    then CONSTRAINED_UNPREDICTABLE;
8  if Rd    == '11x1' || Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
9  if fcond == '111x'                 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <Rd> | Destination general-purpose register. |
| <Rn> | First source general-purpose register (ZR is permitted, PC is not). |
| <Rm> | Second source general-purpose register (ZR is permitted, PC is not). |
| <fcond> | The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values: |

| | | |
|---|---|---|
| EQ | Encoded as | fcond = 0000 |
| NE | Encoded as | fcond = 0001 |
| CS | Encoded as | fcond = 0010 |
| CC | Encoded as | fcond = 0011 |
| MI | Encoded as | fcond = 0100 |
| PL | Encoded as | fcond = 0101 |
| VS | Encoded as | fcond = 0110 |
| VC | Encoded as | fcond = 0111 |
| HI | Encoded as | fcond = 1000 |
| LS | Encoded as | fcond = 1001 |
| GE | Encoded as | fcond = 1010 |
| LT | Encoded as | fcond = 1011 |
| GT | Encoded as | fcond = 1100 |
| LE | Encoded as | fcond = 1101 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2
3  bits(32) result;
```

```
4  if ConditionHolds(fcond) then
5      result = RZ[n];
6  else
7      result = NOT(RZ[m]);
8  R[d] = result;
```

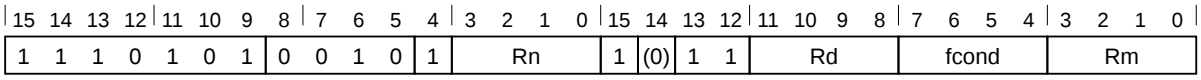
## C2.4.46 CSNEG

Conditional Select Negation. Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register negated.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Main Extension only*




#### T1: CSNEG variant

```
CSNEG Rd, Rn, Rm, <fcond>
```

### Decode for this encoding

```
1  if Rm == '1101'              then SEE "Related encodings";
2  if !HaveMainExt()            then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  d = UInt(Rd);
5  m = UInt(Rm);
6  n = UInt(Rn);
7  if InITBlock()                    then CONSTRAINED_UNPREDICTABLE;
8  if Rd    == '11x1' || Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
9  if fcond == '111x'                 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <Rd> | Destination general-purpose register. |
| <Rn> | First source general-purpose register (ZR is permitted, PC is not). |
| <Rm> | Second source general-purpose register (ZR is permitted, PC is not). |
| <fcond> | The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values: |

| | | |
|---|---|---|
| EQ | Encoded as | fcond = 0000 |
| NE | Encoded as | fcond = 0001 |
| CS | Encoded as | fcond = 0010 |
| CC | Encoded as | fcond = 0011 |
| MI | Encoded as | fcond = 0100 |
| PL | Encoded as | fcond = 0101 |
| VS | Encoded as | fcond = 0110 |
| VC | Encoded as | fcond = 0111 |
| HI | Encoded as | fcond = 1000 |
| LS | Encoded as | fcond = 1001 |
| GE | Encoded as | fcond = 1010 |
| LT | Encoded as | fcond = 1011 |
| GT | Encoded as | fcond = 1100 |
| LE | Encoded as | fcond = 1101 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2
3  bits(32) result;
```

```
4  if ConditionHolds(fcond) then
5      result = RZ[n];
6  else
7      result = NOT(RZ[m]);
8      result = result + 1;
9  R[d] = result;
```

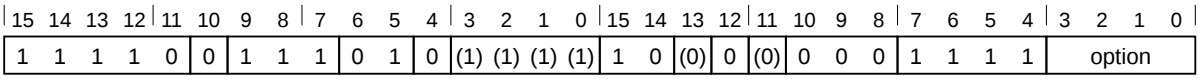
## C2.4.47 DBG

Debug hint. Debug Hint provides a hint to debug trace support and related debug systems. See debug architecture documentation for what use (if any) is made of this instruction.

DBG is a NOP-compatible hint.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
DBG{<c>}{<q>} #<option>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  // Any decoding of 'option' is specified by the debug system
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <option> | Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field. |

### Operation for all encodings

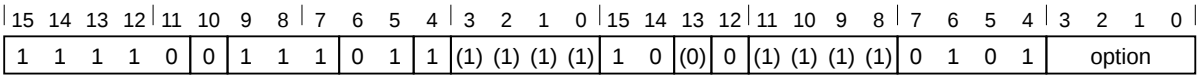
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      Hint_Debug(option);
```

## C2.4.48 DMB

Data Memory Barrier. Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the PE.

### T1

*Armv8-M*




#### T1 variant

`DMB{<c>}{<q>} {<option>}`

### Decode for this encoding

```
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <option> | Specifies an optional limitation on the barrier operation. Values are:<br>SY Full system barrier operation, encoded as option = 0b1111. Can be omitted.<br>All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

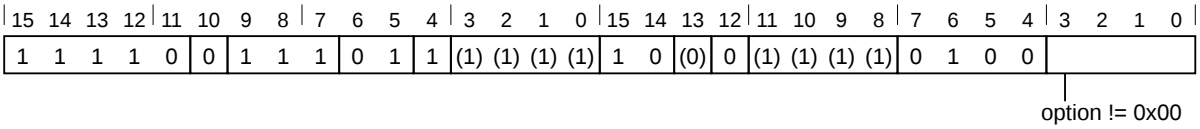
## C2.4.49 DSB

Data Synchronization Barrier. Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes only when both:

- Any explicit memory access made before this instruction is complete.
- The side-effects of any SCS access that performs a context-altering operation are visible.

### T1

*Armv8-M*




#### T1 variant

```
DSB{<c>}{<q>} {<option>}
```

### Decode for this encoding

```
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <option> | Specifies an optional limitation on the barrier operation. Values are:<br>SY Full system barrier operation, encoded as option = 0b1111. Can be omitted.<br>All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software. |

### Operation for all encodings

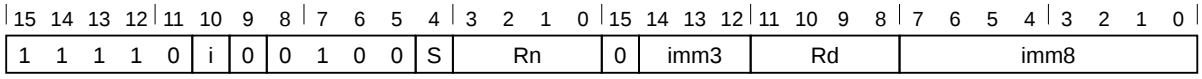
```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

## C2.4.50 EOR (immediate)

Exclusive OR (immediate). Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**EOR variant**

**Applies when `S == 0`.**

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**EORS variant**

**Applies when `S == 1 && Rd != 1111`.**

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
4  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5  if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

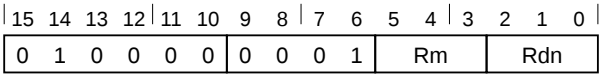
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = R[n] EOR imm32;
4      R[d] = result;
5      if setflags then
6          APSR.N = result<31>;
7          APSR.Z = IsZeroBit(result);
8          APSR.C = carry;
9          // APSR.V unchanged
```

## C2.4.51 EOR (register)

Exclusive OR (register). Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

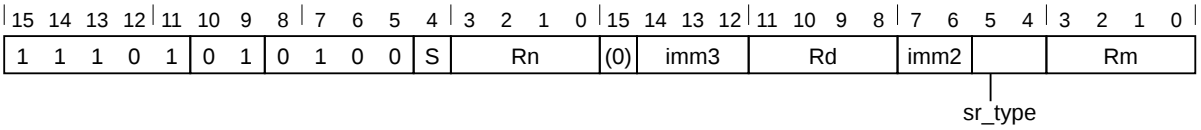
```
EOR<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
EORS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




**EOR, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**EOR, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
EOR<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**EORS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11`.**

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**EORS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111`.**

```
EORS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if Rd == '1111' && S == '1' then SEE "TEQ (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

```
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
```

## C2.4.52 ESB

Error Synchronization Barrier. Error Synchronization Barrier is used to synchronize any asynchronous RAS exceptions. That is, RAS errors notified to the PE will not silently propagate past this instruction.

### T1

*Armv8.1-M Main Extension only*

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

#### T1: ESB variant

```
ESB<c>
```

### Decode for this encoding

```
1
2  if !HaveMainExt() then UNDEFINED;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      supported = boolean IMPLEMENTATION_DEFINED "Detection of Unrecoverable errors supported";
5      if HasArchVersion(Armv8p1) && supported then
6          HandleException(SynchronizeBusFault());
7          // If there is a pending BusFault (which might have been forced to be recognised as a
8          // direct result of the call to SynchronizeBusFault() above) then an ESB also acts as
9          // a Data Synchronization Barrier so that a subsequent load of the memory mapped
               syndrome
10         // registers and pending bits (BFSR, RFSR, and SHCSR) is guaranteed to return the
               correct
11         // values.
12         if SHCSR.BUSFAULTPENDED == '1' then
13             DataSynchronizationBarrier('1111');
```

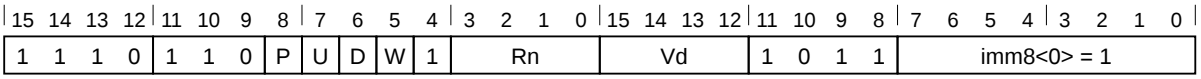
## C2.4.53 FLDMDBX, FLDMIAX

FLDMX (Decrement Before, Increment After). FLDMX (Decrement Before, Increment After) loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

Arm deprecates use of FLDMDBX and FLDMIAX, except for disassembly purposes, and reassembly of disassembled code.

### T1

*Armv8-M Floating-point Extension only*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
FLDMDBX{<c>}{<q>} <Rn>{!}, <dreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1`.**

```
FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

### Decode for this encoding

```
1  if P == '0' && U == '0' && W == '0'                then SEE "Related encodings";
2  if P == '0' && U == '1' && W == '0' && Rn == '1111' then SEE "VSCCLRM";
3  if P == '1' && W == '0'                            then SEE VLDR;
4  CheckDecodeFaults(ExtType_MveOrFp);
5  if P == U && W == '1' then UNDEFINED;
6  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
7  single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
8  d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
9  regs = UInt(imm8) DIV 2;
10 if n == 15 then UNPREDICTABLE;
11 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `FLDMX` with the same addressing mode but loads no registers.

#### CONSTRAINED UNPREDICTABLE behavior

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

### CONSTRAINED UNPREDICTABLE behavior

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0. |
| `<dreglist>` | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    address = if add then R[n]       else R[n]-imm32;
    regval  = if add then R[n]+imm32 else R[n]-imm32;

    // Determine if the stack pointer limit must be checked
    if n == 13 && wback then
        // If memory operation is not performed as a result of a stack limit violation,
        // and the write-back of the SP itself does not raise a stack limit violation, it
        // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
        // Arm recommends that any instruction which discards a memory access as
        // a result of a stack limit violation, and where the write-back of the SP itself
        // does not raise a stack limit violation, generates an SPLIM exception.
        if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
            if ViolatesSPLim(LookUpSP(), address) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to
                    HardFault
                excInfo = CreateException(UsageFault);
                HandleException(excInfo);
        applylimit = TRUE;
    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !(applylimit && ViolatesSPLim(LookUpSP(), regval)) then
        for r = 0 to regs-1
            if single_regs then
                S[d+r]  = MemA[address,4];
                address = address+4;
            else
                if (d+r) < 16 || !VFPSmallRegisterBank() then
                    word1  = MemA[address,4];  word2 = MemA[address+4,4];
                    // Combine the word-aligned words in the correct order for
                    // current endianness.
                    D[d+r] = if BigEndian(address, 8) then word1:word2 else word2:word1;
                elsif boolean UNKNOWN then
                    - = MemA[address,4]; - = MemA[address+4,4];
                address = address+8;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

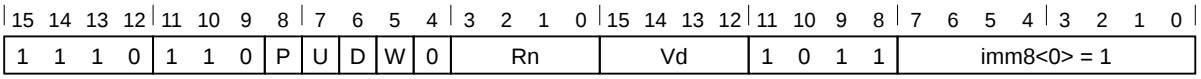
## C2.4.54 FSTMDBX, FSTMIAX

FSTMX (Decrement Before, Increment After). FSTMX (Decrement Before, Increment After) stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

### T1

*Armv8-M Floating-point Extension only*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
FSTMDBX{<c>}{<q>} <Rn>{!}, <dreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1`.**

```
FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>
```

### Decode for this encoding

```
1  if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2  if P == '1' && W == '0' then SEE VSTR;
3  CheckDecodeFaults(ExtType_MveOrFp);
4  if P == U && W == '1' then UNDEFINED;
5  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6  single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
7  d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
8  regs = UInt(imm8) DIV 2;
9  if n == 15 then UNPREDICTABLE;
10 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `FSTMX` with the same addressing mode but stores no registers.

#### CONSTRAINED UNPREDICTABLE behavior

If `regs > 16 || (d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### CONSTRAINED UNPREDICTABLE behavior

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0. |
| `<dreglist>` | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list plus one. The list must contain at least one register, all registers must be in the range D0-D15, and must not contain more than 16 registers. |

## Operation for all encodings

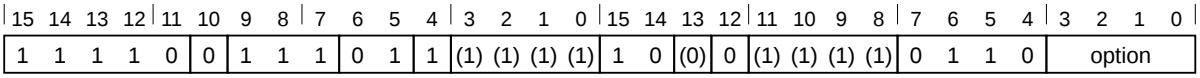
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    address = if add then R[n]       else R[n]-imm32;
    regval  = if add then R[n]+imm32 else R[n]-imm32;

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), regval);
    else
        violatesLimit = FALSE;

    // Memory operation only performed if limit not violated
    if !violatesLimit then
        for r = 0 to regs-1
            if single_regs then
                MemA[address,4] = S[d+r];
                address         = address+4;
            else
                // Store as two word-aligned words in the correct order for current
                    endianness.
                if (d+r) < 16 || !VFPSmallRegisterBank() then
                    bigEndian         = BigEndian(address, 8);
                    MemA[address,4]   = if bigEndian then D[d+r]<63:32> else D[d+r]<31:0>;
                    MemA[address+4,4] = if bigEndian then D[d+r]<31:0>  else D[d+r]<63:32>;
                elsif boolean UNKNOWN then
                    MemA[address,4]   = bits(32) UNKNOWN;
                    MemA[address+4,4] = bits(32) UNKNOWN;
                address = address+8;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

## C2.4.55 ISB

Instruction Synchronization Barrier. Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event.

### T1

*Armv8-M*




#### T1 variant

```
ISB{<c>}{<q>} {<option>}
```

### Decode for this encoding

```
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <option> | Specifies an optional limitation on the barrier operation. Values are:<br>SY Full system barrier operation, encoded as option = 0b1111. Can be omitted.<br>All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

## C2.4.56 IT

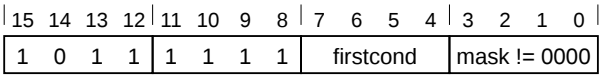
If-Then. If Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

IT does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than CMP (register), CMN (register), and TST (register), do not set the condition code flags. The `AL` condition can be specified to get this changed behavior without conditional execution.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`IT{<x>{<y>{<z>}}}{<q>} <cond>`

### Decode for this encoding

```
1  if mask == '0000' then SEE "Related encodings";
2  if !HaveMainExt() then UNDEFINED;
3  if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
4  if InITBlock() then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The ‘1111’ condition is treated as being the same as the ‘1110’ condition, meaning always, and the ITSTATE state machine is progressed in the same way as for any other cond_base value.

### Assembler symbols for all encodings

| | |
|---|---|
| `<x>` | The condition for the second instruction in the IT block. If omitted, the "mask" field is set to `0b1000`. If present it is encoded in the "mask[3]" field:<br>`E` `NOT firstcond[0]`<br>`T` `firstcond[0]` |
| `<y>` | The condition for the third instruction in the IT block. If omitted and `<x>` is present, the "mask[2:0]" field is set to `0b100`. If `<y>` is present it is encoded in the "mask[2]" field:<br>`E` `NOT firstcond[0]`<br>`T` `firstcond[0]` |
| `<z>` | The condition for the fourth instruction in the IT block. If omitted and `<y>` is present, the "mask[1:0]" field is set to `0b10`. If `<z>` is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:<br>`E` `NOT firstcond[0]`<br>`T` `firstcond[0]` |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<cond>` | The condition for the first instruction in the IT block, encoded in the "firstcond" field. See C1.3 *Conditional execution* on page 429 for the range of conditions available, and the encodings. |

## Operation for all encodings

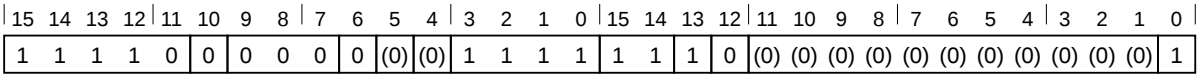
```
1 EncodingSpecificOperations();
2 ITSTATE<7:0> = firstcond:mask;
```

## C2.4.57 LCTP

Loop Clear with Tail Predication. Exits loop mode by invalidating LO_BRANCH_INFO and clears any tail predication being applied.

### T1

*Armv8.1-M MVE*




#### T1: LCTP variant

LCTP<c>

### Decode for this encoding

```
1  if !HaveLOBExt() then UNDEFINED;
2  if !HaveMve()    then UNDEFINED;
3  HandleException(CheckCPEnabled(10));
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

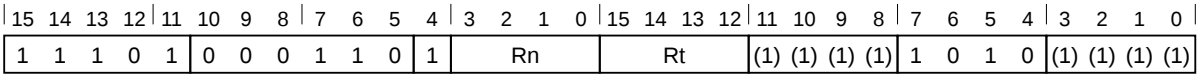
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      ExecuteFPCheck();
5      FPSCR.LTPSIZE = 4<2:0>; // Disable loop predication
6      if LO_BRANCH_INFO.BF == '0' then
7          LO_BRANCH_INFO.VALID = '0';
```

## C2.4.58 LDA

Load-Acquire Word. Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt); n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

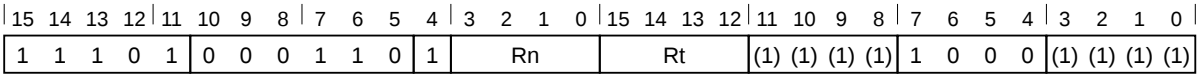
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      R[t] = MemO[address, 4];
```

## C2.4.59 LDAB

Load-Acquire Byte. Load-Acquire Byte loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt); n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      R[t] = ZeroExtend(MemO[address, 1], 32);
```

## C2.4.60 LDAEX

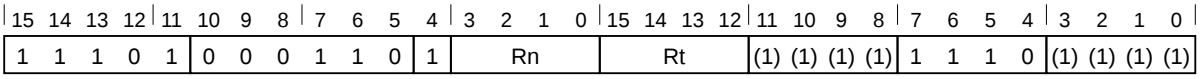
Load-Acquire Exclusive Word. Load-Acquire Exclusive Word loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      SetExclusiveMonitors(address, 4);
5      R[t] = MemO[address, 4];
```

## C2.4.61 LDAEXB

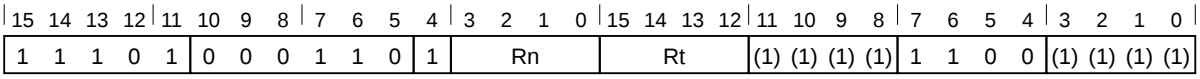
Load-Acquire Exclusive Byte. Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      SetExclusiveMonitors(address, 1);
5      R[t] = ZeroExtend(MemO[address, 1], 32);
```

## C2.4.62 LDAEXH

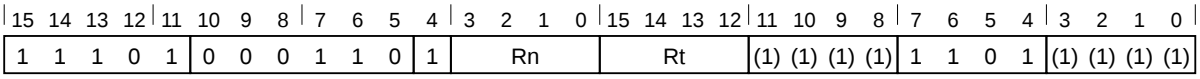
Load-Acquire Exclusive Halfword. Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

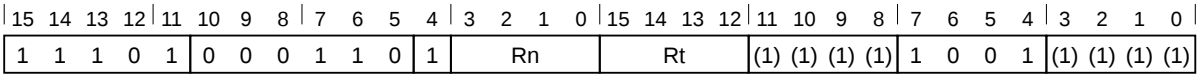
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      SetExclusiveMonitors(address, 2);
5      R[t] = ZeroExtend(MemO[address, 2], 32);
```

## C2.4.63 LDAH

Load-Acquire Halfword. Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      R[t] = ZeroExtend(MemO[address, 2], 32);
```

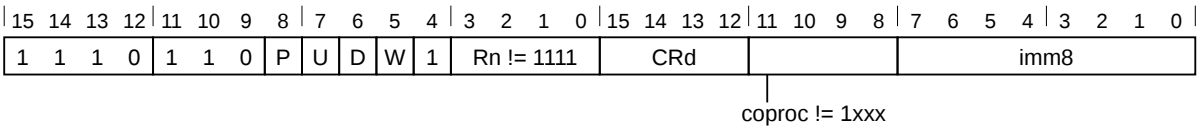
## C2.4.64 LDC, LDC2 (immediate)

Load Coprocessor (immediate). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

### T1

*Armv8-M Main Extension only*




**Offset variant**

**Applies when `P == 1 && W == 0`.**

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

**Unindexed variant**

**Applies when `P == 0 && U == 1 && W == 0`.**

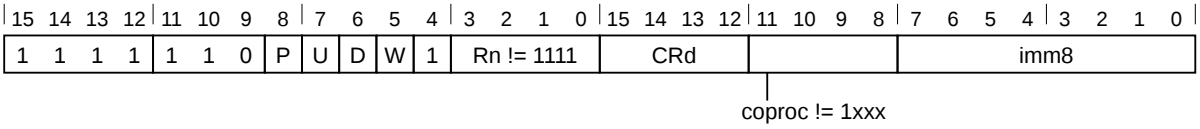
```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDC (literal)";
2  if coproc IN {'100x', '101x', '111x'}           then SEE "Floating-point and MVE";
3  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
4  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt() then UNDEFINED;
6  n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

### T2

*Armv8-M Main Extension only*



#### Offset variant

**Applies when `P == 1 && W == 0`.**

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

**Applies when `P == 1 && W == 1`.**

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### Unindexed variant

**Applies when `P == 0 && U == 1 && W == 0`.**

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

### Decode for this encoding

```
if Rn == '1111'                       then SEE "LDC (literal)";
if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

### Assembler symbols for all encodings

| | |
|---|---|
| `L` | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<coproc>` | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| `<CRd>` | Is the coprocessor register to be transferred, encoded in the "CRd" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see C2.4.65 *LDC, LDC2 (literal)* on page 598. |
| `<option>` | Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>– when `U = 0`<br>+ when `U = 1` |
| `<imm>` | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);

    thisInstr = ThisInstr();
    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];

        // Determine if the stack pointer limit check should be performed
        if wback && n == 13 then
            violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
        else
            violatesLimit = FALSE;

        // Memory operation only performed if limit not violated
        if !violatesLimit then
            repeat
                Coproc_SendLoadedWord(MemA[address,4], cp, thisInstr);
                address = address + 4;
            until Coproc_DoneLoading(cp, thisInstr);

        // If the stack pointer is being updated a fault will be raised
        // if the limit is violated
        if wback then RSPCheck[n] = offset_addr;
```

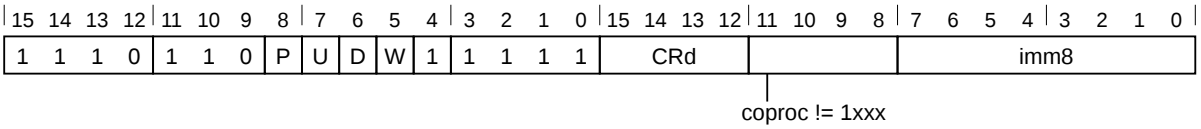
## C2.4.65 LDC, LDC2 (literal)

Load Coprocessor (literal). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

### T1

*Armv8-M Main Extension only*




#### T1 variant

**Applies when `!(P == 0 && U == 0 && W == 0)`.**

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}           then SEE "Floating-point and MVE";
2  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
3  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
4  if !HaveMainExt() then UNDEFINED;
5  index = (P == '1'); // Always TRUE in the T32 instruction set
6  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
7  if W == '1' || P == '0' then UNPREDICTABLE;
```

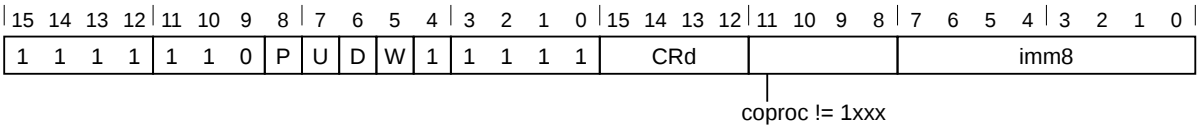
#### CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes as `LDC` with writeback to the PC.

### T2

*Armv8-M Main Extension only*




#### T2 variant

**Applies when `!(P == 0 && U == 0 && W == 0)`.**

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

## Decode for this encoding

```
if coproc IN {'100x', '101x', '111x'}           then SEE "Floating-point and MVE";
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
index = (P == '1'); // Always TRUE in the T32 instruction set
add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || P == '0' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction executes as `LDC` with writeback to the PC.

## Assembler symbols for all encodings

| | |
|---|---|
| `L` | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<coproc>` | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| `<CRd>` | Is the coprocessor register to be transferred, encoded in the "CRd" field. |
| `<label>` | The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the Align `(PC, 4)` value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE` (encoded as U == 1). If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE (encoded as U == 0)`. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>– when `U = 0`<br>+ when `U = 1` |
| `<imm>` | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);

    thisInstr = ThisInstr();
    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);
```

## C2.4.66 LDM, LDMIA, LDMFD

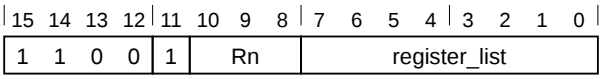
Load Multiple (Increment After, Full Descending). Load Multiple loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as a branch address, a function return value, or an exception return value. Bit[0] of the address in the PC complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] of the target address is 0, and the target address is not FNC_RETURN or EXC_RETURN, the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

This instruction is used by the alias POP (multiple registers).

### T1

*Armv8-M*




#### T1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers>
  // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers>
  // Alternate syntax, Full Descending stack
```

### Decode for this encoding

```
1 n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
2 if BitCount(registers) < 1 then UNPREDICTABLE;
```

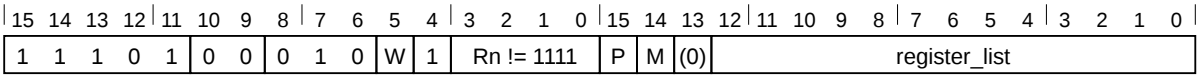
#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDM{IA}{<c>}.W <Rn>{!}, <registers>
  // Preferred syntax
  // if <Rn>, '!' and <registers> can be represented in T1
LDMFD{<c>}.W <Rn>{!}, <registers>
```

```
  // Alternate syntax
  // Full Descending stack, if <Rn>, '!' and <registers> can be represented in T1
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers>
  // Preferred syntax
LDMFD{<c>}{<q>} <Rn>{!}, <registers>
  // Alternate syntax, Full Descending stack
```

## Decode for this encoding

```
1  if Rn == '1111' && HasArchVersion(Armv8p1) then SEE "CLRM";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
4  if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
5  if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
6  if wback && registers<n> == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

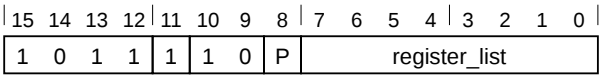
### CONSTRAINED UNPREDICTABLE behavior

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

### T3

*Armv8-M*




#### T3 variant

`LDM{<c>}{<q>} SP!, <registers>`

### Decode for this encoding

```
1  n = 13; wback = TRUE;
2  registers = P:'0000000':register_list;
3  if BitCount(registers) < 1 then UNPREDICTABLE;
4  if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

### Assembler symbols for all encodings

| | |
|---|---|
| `IA` | Is an optional suffix for the Increment After form. |
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `!` | For encoding T1: the address adjusted by the size of the data loaded is written back to the base register. It is omitted if `<Rn>` is included in `<registers>`, otherwise it must be present. For encoding T2: the address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| `<registers>` | For encoding T1: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. For encoding T2: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:<br>- The LR must not be in the list.<br>- The instruction must be either outside any IT block, or the last instruction in an IT block.<br>For encoding T3: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
```

```
    address = R[n];
    if n == 13 && wback then
        // If memory operation is not performed as a result of a stack limit violation,
        // and the write-back of the SP itself does not raise a stack limit violation, it
        // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
        // Arm recommends that any instruction which discards a memory access as
        // a result of a stack limit violation, and where the write-back of the SP itself
        // does not raise a stack limit violation, generates an SPLIM exception.
        if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
            if ViolatesSPLim(LookUpSP(), address) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to
                    HardFault
                excInfo = CreateException(UsageFault);
                HandleException(excInfo);
        applylimit = TRUE;
    else
        applylimit = FALSE;

    for i = 0 to 14
        // If R[n] is the SP, memory operation only performed if limit not violated
        if registers<i> == '1' && !(applylimit && ViolatesSPLim(LookUpSP(), address)) then
            if i != n then
                R[i]       = MemA[address,4];
            else
                newBaseVal = MemA[address,4];
            address = address + 4;
    if registers<15> == '1' && !(applylimit && ViolatesSPLim(LookUpSP(), address)) then
        newPCVal = MemA[address,4];

    // If the register list contains the register that holds the base address it
    // must be updated after all memory reads have been performed. This prevents
    // the base address being overwritten if one of the memory reads generates a
    // fault.
    if registers<n> == '1' then
        wback      = TRUE;
    else
        newBaseVal = R[n] + 4*BitCount(registers);
    // If the PC is in the register list update that now, which might raise a fault
    // Likewise if R[n] is the SP writing back might raise a fault due to SP limit violation
    if registers<15> == '1' then
        LoadWritePC(newPCVal, n, newBaseVal, wback, FALSE);
    elsif wback then
        RSPCheck[n] = newBaseVal;
```

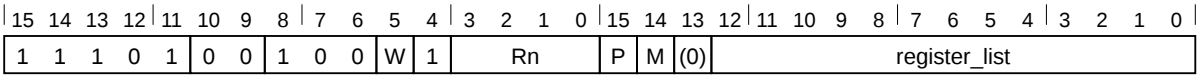
## C2.4.67 LDMDB, LDMEA

Load Multiple Decrement Before (Empty Ascending). Load Multiple Decrement Before (Load Multiple Empty Ascending) loads multiple registers from sequential memory locations using an address from a base register. The sequential memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers>
  // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers>
  // Alternate syntax, Empty Ascending stack
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
3  if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
4  if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
5  if wback && registers<n> == '1' then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as `LDM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers loaded.

#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads a single register using the specified addressing modes.
- The instruction executes as LDM with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

### CONSTRAINED UNPREDICTABLE behavior

If P == '1' && M == '1', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| <registers> | Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. If the PC is in the list:<br>- The LR must not be in the list.<br>- The instruction must be either outside any IT block, or the last instruction in an IT block. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback && registers<n> == '0' then
        violatesLimit = ViolatesSPLim(LookUpSP(), address);
    else
        violatesLimit = FALSE;

    for i = 0 to 15
        // Memory operation only performed if limit not violated
        if registers<i> == '1' && !violatesLimit then
            data    = MemA[address,4];
            address = address + 4;
            if i == 15 then
                newPCVal   = data;
            elsif i == n then
                newBaseVal = data;
            else
                R[i]       = data;

    // If the register list contains the register that holds the base address it
    // must be updated after all memory reads have been performed. This prevents
    // the base address being overwritten if one of the memory reads generates a
    // fault.
    if registers<n> == '1' then
        wback      = TRUE;
```

```
29      else
30          newBaseVal = R[n] - 4*BitCount(registers);
31      // If the PC is in the register list update that now, which may raise a fault
32      if registers<15> == '1' then
33          LoadWritePC(newPCVal, n, newBaseVal, wback, TRUE);
34      elsif wback then
35          RSPCheck[n] = newBaseVal;
```

## C2.4.68 LDR (immediate)

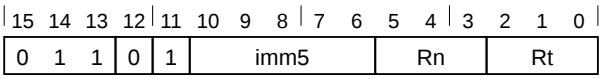
Load Register (immediate). Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

This instruction is used by the alias POP (single register).

### T1

*Armv8-M*




#### T1 variant

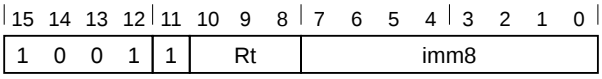
```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M*




#### T2 variant

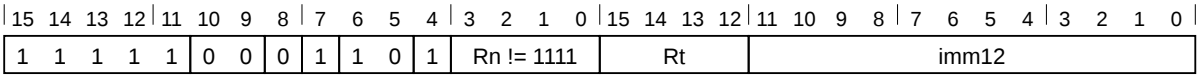
```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

### Decode for this encoding

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

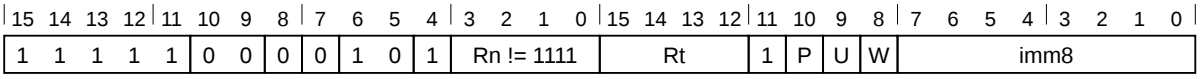
```
LDR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1 or T2
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32); index = TRUE;  add = TRUE;
wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T4

*Armv8-M Main Extension only*




**Offset variant**

**Applies when `P == 1 && U == 0 && W == 0`.**

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
LDR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
LDR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
if P == '1' && U == '1' && W == '0' then SEE LDRT;
if P == '0' && W == '0' then UNDEFINED;
if !HaveMainExt() 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;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Alias conditions

| Alias | preferred when |
|---|---|
| POP (single register) | `Rn == '1101' && U == '1' && imm8 == '00000100'` |

## Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <Rt> | For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.<br>For encoding T3: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC.<br>For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. |
| <Rn> | For encoding T1: is the general-purpose base register, encoded in the "Rn" field.<br>For encoding T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.69 *LDR (literal)* on page 611. |
| +/− | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>− when `U == 0`<br>+ when `U == 1` |
| + | Specifies the offset is added to the base register. |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.<br>For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.<br>For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T4: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
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];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        data = MemU[address,4];

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data, n, offset_addr, wback, TRUE);
        else
            UNPREDICTABLE;
    else
        if wback then RSPCheck[n] = offset_addr;
        R[t] = data;
```

### CONSTRAINED UNPREDICTABLE behavior

If `t == 15 && address<1:0> != '00'`, then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.

- The instruction generates an UNALIGNED UsageFault.

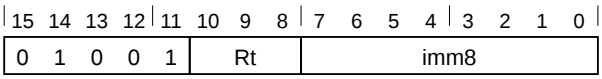
## C2.4.69 LDR (literal)

Load Register (literal). Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

### T1

*Armv8-M*




#### T1 variant

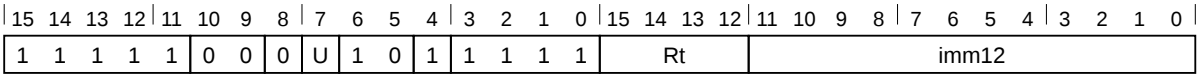
```
LDR{<c>}{<q>} <Rt>, <label>
  // Normal form
```

### Decode for this encoding

```
1  t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDR{<c>}.W <Rt>, <label>
  // Preferred syntax, and <Rt>, <label> can be represented in T1
LDR{<c>}{<q>} <Rt>, <label>
  // Preferred syntax
LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
3  if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field. For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. |

| | |
|---|---|
| <label> | For encoding T1: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are Multiples of four in the range 0 to 1020.<br>For encoding T2: the label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

## Operation for all encodings

```
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, 0, Zeros(32), FALSE, FALSE);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

### CONSTRAINED UNPREDICTABLE behavior

If t == 15 && address<1:0> != '00', then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

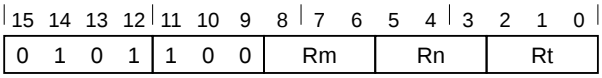
## C2.4.70 LDR (register)

Load Register (register). Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

### T1

*Armv8-M*




#### T1 variant

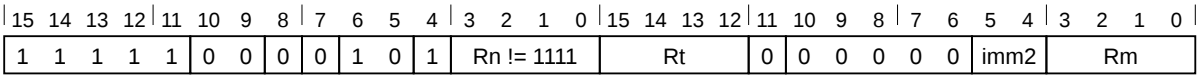
```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
if !HaveMainExt() 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 m IN {13,15} then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | For encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field. For encoding T2: is the general-purpose register to be transferred, encoded in the "Rt" field. The SP can be used. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. |

| | |
|---|---|
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

## Operation for all encodings

```
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];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        data = MemU[address,4];

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data, n, offset_addr, wback, TRUE);
        else
            UNPREDICTABLE;
    else
        if wback then RSPCheck[n] = offset_addr;
        R[t] = data;
```

### CONSTRAINED UNPREDICTABLE behavior

If `t == 15 && address<1:0> != '00'`, then one of the following behaviors must occur:

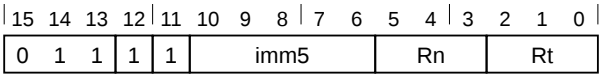
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

## C2.4.71 LDRB (immediate)

Load Register Byte (immediate). Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M*




#### T1 variant

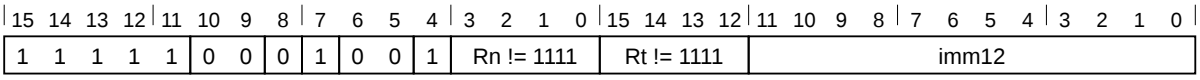
```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

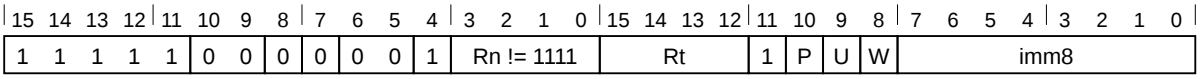
```
LDRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
if Rt == '1111' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRB (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.**

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
LDRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
1  if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD (immediate)";
2  if Rn == '1111' then SEE "LDRB (literal)";
3  if P == '1' && U == '1' && W == '0' then SEE LDRBT;
4  if P == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt() then UNDEFINED;
6  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if t == 13 || (wback && n == t) then UNPREDICTABLE;
9  if t == 15 && W == '1' then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding T1: is the general-purpose base register, encoded in the "Rn" field.<br>For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.72 *LDRB (literal)* on page 618. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U = 0`<br>+ when `U = 1` |
| + | Specifies the offset is added to the base register. |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4      address = if index then offset_addr else R[n];
```

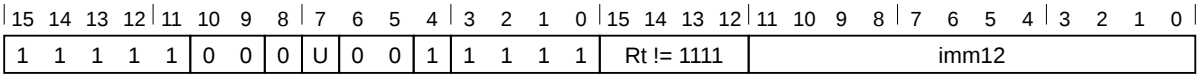
```
5
6       // Determine if the stack pointer limit should be checked
7       if n == 13 && wback then
8           violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
9       else
10          violatesLimit = FALSE;
11
12      // Memory operation only performed if limit not violated
13      if !violatesLimit then
14          R[t] = ZeroExtend(MemU[address,1], 32);
15
16      // If the stack pointer is being updated a fault will be raised if
17      // the limit is violated
18      if wback then RSPCheck[n] = offset_addr;
```

## C2.4.72 LDRB (literal)

Load Register Byte (literal). Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRB{<c>}{<q>} <Rt>, <label>
  // Preferred syntax
LDRB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if Rt == '1111' then SEE "PLD (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
4  if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

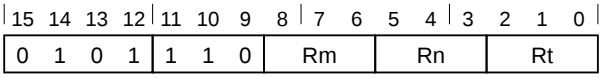
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      base = Align(PC,4);
4      address = if add then (base + imm32) else (base - imm32);
5      R[t] = ZeroExtend(MemU[address,1], 32);
```

## C2.4.73 LDRB (register)

Load Register Byte (register). Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

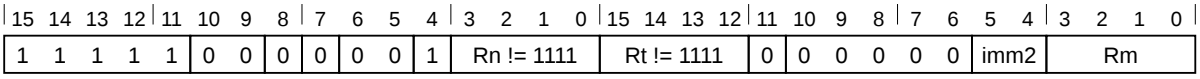
```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rt == '1111' then SEE "PLD (register)";
if Rn == '1111' then SEE "LDRB (literal)";
if !HaveMainExt() 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 == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
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);
```

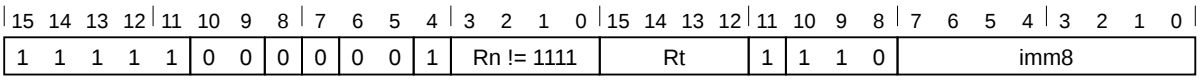
## C2.4.74 LDRBT

Load Register Byte Unprivileged. 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.

When privileged software uses an LDRBT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRB (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

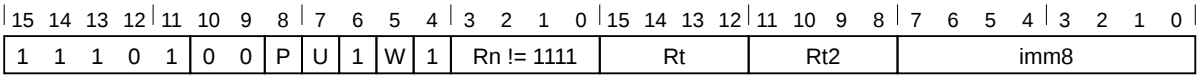
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      R[t] = ZeroExtend(MemU_unpriv[address,1],32);
```

## C2.4.75 LDRD (immediate)

Load Register Dual (immediate). Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M Main Extension only*




**Offset variant**

**Applies when `P == 1 && W == 0`.**

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

### Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  if Rn == '1111' then SEE "LDRD (literal)";
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
5  index = (P == '1');  add = (U == '1');  wback = (W == '1');
6  if wback && (n == t || n == t2) then UNPREDICTABLE;
7  if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

#### CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.76 *LDRD (literal)* on page 624. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | For the offset variant: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.<br>For the post-indexed and pre-indexed variant: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm>/4. |

### Operation for all encodings

```
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];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        R[t]  = MemA[address,4];
        R[t2] = MemA[address+4,4];

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

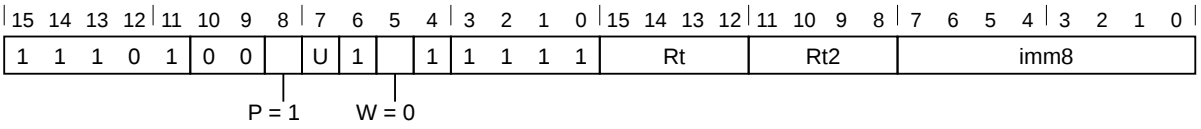
## C2.4.76 LDRD (literal)

Load Register Dual (literal). Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers.

For the M profile, the PC value must be word-aligned, otherwise the behavior of the instruction is UNPREDICTABLE.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label>
  // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>]
  // Alternative form
```

### Decode for this encoding

```
1  if P == '0' && W == '0'             then SEE "Related encodings";
2  if P == '1' && W == '1' && U == '0' then SEE SG;
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  t2 = UInt(Rt2);
5  imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
6  if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
7  if W == '1' then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The load instruction executes but the destination register takes an UNKNOWN value.

#### CONSTRAINED UNPREDICTABLE behavior

If `W == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes without writeback of the base address.
- The instruction uses post-indexed addressing when P == ‘0’ and uses pre-indexed addressing otherwise.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rt2>` | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |

| | |
|---|---|
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is the optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
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];
```

### CONSTRAINED UNPREDICTABLE behavior

If PC<1:0> != '00', then one of the following behaviors must occur:

- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

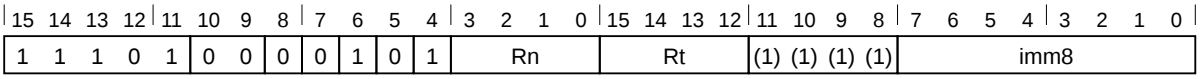
## C2.4.77 LDREX

Load Register Exclusive. Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

### T1

*Armv8-M*




#### T1 variant

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| <imm> | The immediate offset added to the value of <Rn> to calculate the address. <imm> can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      SetExclusiveMonitors(address,4);
5      R[t] = MemA[address,4];
```

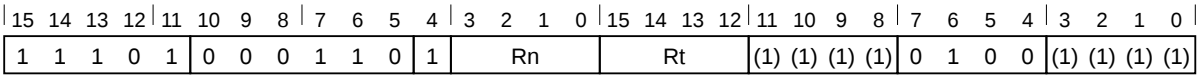
## C2.4.78 LDREXB

Load Register Exclusive Byte. Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

### T1

*Armv8-M*




#### T1 variant

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      SetExclusiveMonitors(address,1);
5      R[t] = ZeroExtend(MemA[address,1], 32);
```

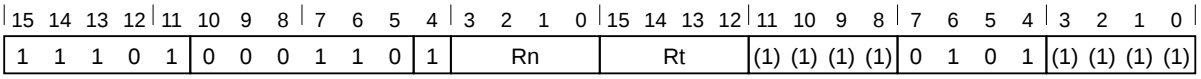
## C2.4.79 LDREXH

Load Register Exclusive Halfword. Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

### T1

*Armv8-M*




#### T1 variant

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

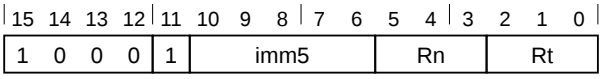
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      SetExclusiveMonitors(address,2);
5      R[t] = ZeroExtend(MemA[address,2], 32);
```

## C2.4.80 LDRH (immediate)

Load Register Halfword (immediate). Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M*




#### T1 variant

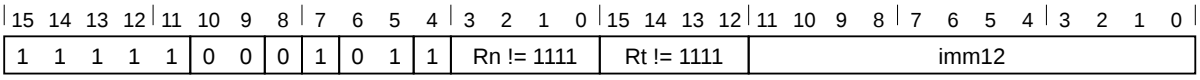
```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
2  index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

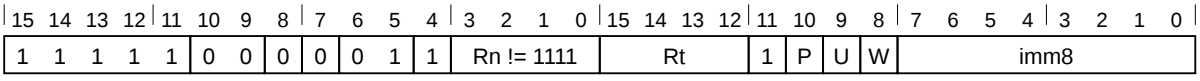
```
LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rt == '1111' then SEE "Related encodings";
2  if Rn == '1111' then SEE "LDRH (literal)";
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
5  index = TRUE;  add = TRUE;  wback = FALSE;
6  if t == 13 then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.**

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
LDRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRH (literal)";
2  if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
3  if P == '1' && U == '1' && W == '0' then SEE LDRHT;
4  if P == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt() then UNDEFINED;
6  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | For encoding T1: is the general-purpose base register, encoded in the "Rn" field.<br>For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.81 *LDRH (literal)* on page 632. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U = 0`<br>+ when `U = 1` |
| + | Specifies the offset is added to the base register. |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
4      address = if index then offset_addr else R[n];
5
```

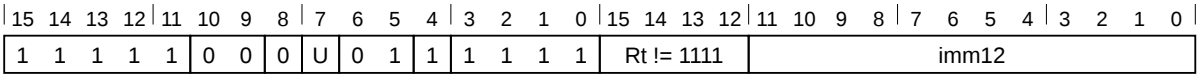
```
    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        R[t] = ZeroExtend(MemU[address,2], 32);

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

## C2.4.81 LDRH (literal)

Load Register Halfword (literal). Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRH{<c>}{<q>} <Rt>, <label>
  // Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if Rt == '1111' then SEE "PLD (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
4  if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

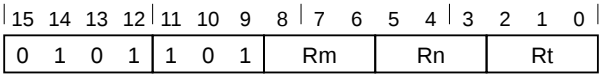
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      base = Align(PC,4);
4      address = if add then (base + imm32) else (base - imm32);
5      data = MemU[address,2];
6      R[t] = ZeroExtend(data, 32);
```

## C2.4.82 LDRH (register)

Load Register Halfword (register). Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

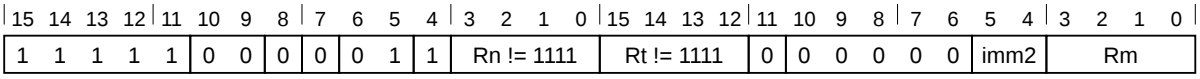
```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "Related encodings";
if !HaveMainExt() 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 == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
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);
```

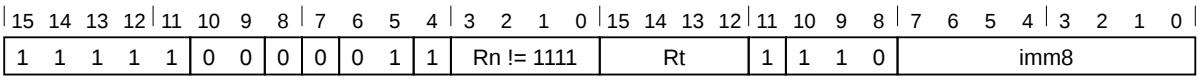
## C2.4.83 LDRHT

Load Register Halfword Unprivileged. Load Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

When privileged software uses an LDRHT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRH (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

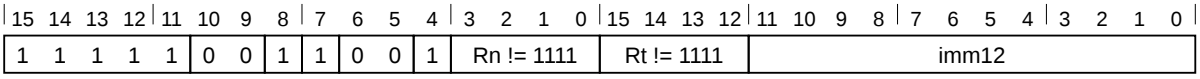
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      data = MemU_unpriv[address,2];
5      R[t] = ZeroExtend(data, 32);
```

## C2.4.84 LDRSB (immediate)

Load Register Signed Byte (immediate). Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M Main Extension only*




#### T1 variant

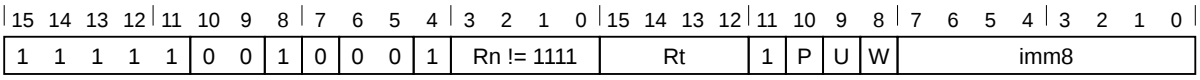
```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rt == '1111' then SEE "PLI (immediate, literal)";
2  if Rn == '1111' then SEE "LDRSB (literal)";
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
5  index = TRUE;  add = TRUE;  wback = FALSE;
6  if t == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `P == 1 && U == 0 && W == 0`.**

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

**Applies when `P == 1 && W == 1`.**

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

### Decode for this encoding

```
1  if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI (immediate, literal)";
2  if Rn == '1111' then SEE "LDRSB (literal)";
3  if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
4  if P == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt() then UNDEFINED;
6  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.85 *LDRSB (literal)* on page 638. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U = 0`<br>+ when `U = 1` |
| `+` | Specifies the offset is added to the base register. |
| `<imm>` | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

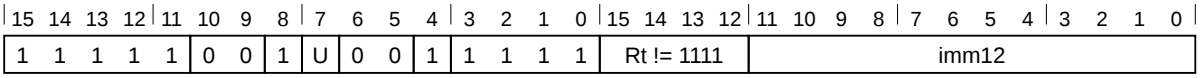
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        R[t] = SignExtend(MemU[address,1], 32);

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

## C2.4.85 LDRSB (literal)

Load Register Signed Byte (literal). Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRSB{<c>}{<q>} <Rt>, <label>
  // Preferred syntax
LDRSB{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if Rt == '1111' then SEE "PLI (immediate, literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
4  if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

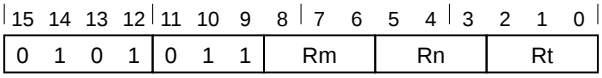
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      base = Align(PC,4);
4      address = if add then (base + imm32) else (base - imm32);
5      R[t] = SignExtend(MemU[address,1], 32);
```

## C2.4.86 LDRSB (register)

Load Register Signed Byte (register). Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

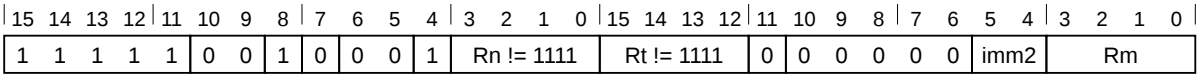
```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rt == '1111' then SEE "PLI (register)";
if Rn == '1111' then SEE "LDRSB (literal)";
if !HaveMainExt() 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 == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
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);
```

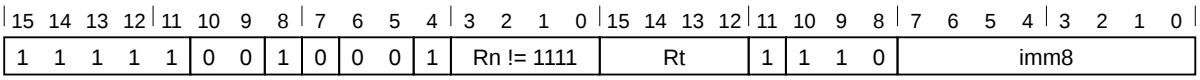
## C2.4.87 LDRSBT

Load Register Signed Byte Unprivileged. 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.

When privileged software uses an LDRSBT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRSB (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

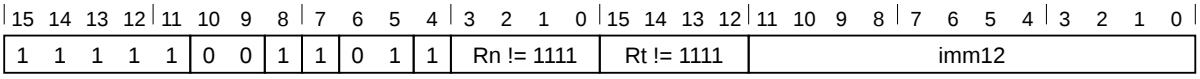
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      R[t] = SignExtend(MemU_unpriv[address,1], 32);
```

## C2.4.88 LDRSH (immediate)

Load Register Signed Halfword (immediate). Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M Main Extension only*




#### T1 variant

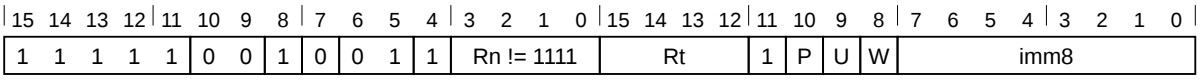
```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRSH (literal)";
2  if Rt == '1111' then SEE "Related encodings";
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
5  index = TRUE;  add = TRUE;  wback = FALSE;
6  if t == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `Rt != 1111 && P == 1 && U == 0 && W == 0`.**

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

**Applies when `P == 1 && W == 1`.**

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRSH (literal)";
2  if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
3  if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
4  if P == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt() then UNDEFINED;
6  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if t == 13 || (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs all of the loads using the specified addressing mode and the content of the register that is written back is UNKNOWN. In addition, if an exception occurs during such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. For PC use see C2.4.89 *LDRSH (literal)* on page 644. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U = 0`<br>+ when `U = 1` |
| `+` | Specifies the offset is added to the base register. |
| `<imm>` | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

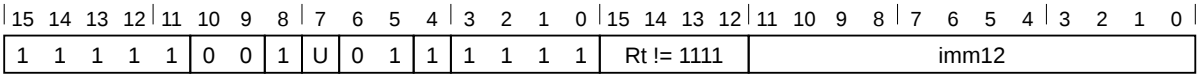
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        R[t] = SignExtend(MemU[address,2], 32);

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

## C2.4.89 LDRSH (literal)

Load Register Signed Halfword (literal). Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRSH{<c>}{<q>} <Rt>, <label>
  // Preferred syntax
LDRSH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1 if Rt == '1111' then SEE "Related encodings";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
4 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <label> | The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE, encoded as U == 1. If the offset is negative, imm32 is equal to minus the offset and add == FALSE, encoded as U == 0. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

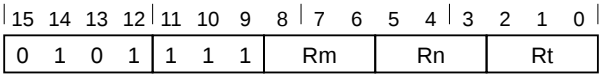
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     base = Align(PC,4);
4     address = if add then (base + imm32) else (base - imm32);
5     data = MemU[address,2];
6     R[t] = SignExtend(data, 32);
```

## C2.4.90 LDRSH (register)

Load Register Signed Halfword (register). Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

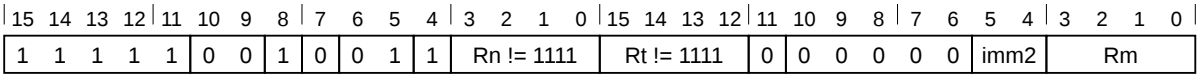
```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related encodings";
if !HaveMainExt() 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 == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
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);
```

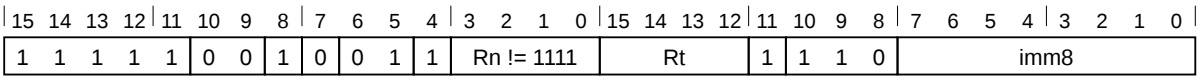
## C2.4.91 LDRSHT

Load Register Signed Halfword Unprivileged. 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.

When privileged software uses an LDRSHT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDRSH (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      data = MemU_unpriv[address,2];
5      R[t] = SignExtend(data, 32);
```

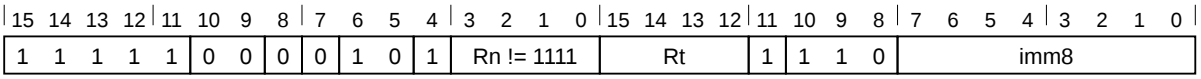
## C2.4.92 LDRT

Load Register Unprivileged. Load Register Unprivileged calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register.

When privileged software uses an LDRT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
LDRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "LDR (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      data = MemU_unpriv[address,4];
5      R[t] = data;
```

## C2.4.93 LE, LETP

Loop End, Loop End with Tail Predication. If additional iterations of a loop are required this instruction branches back to the `<label>`. It also stores the loop information in the loop info cache so that future iterations of the loop will branch back to the start just before the LE instruction is encountered. The first variant of the instruction checks a loop iteration counter (stored in LR) to determine if additional iterations are required. It also decrements the counter ready for the next iteration.

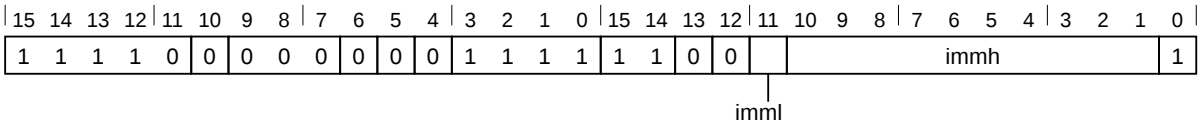
The second variant does not use an iteration count and always triggers another iteration of the loop.

The third (TP) variant also checks the loop iteration counter to determine if additional iterations are required. However the counter is decremented by the number of elements in a vector (as indicated by the FPSCR.LTPSIZE field). On the last iteration of the loop, this variant disables tail predication.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Low Overhead Branch Extension*




#### T1: LE variant

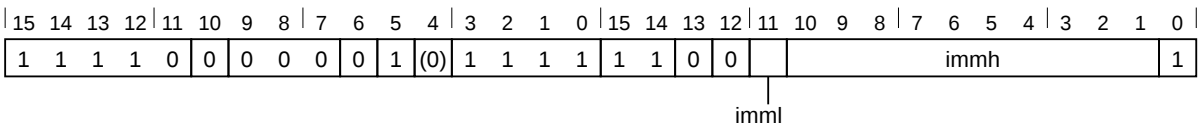
```
LE LR, <label>
```

### Decode for this encoding

```
if !HaveLOBExt() then UNDEFINED;

forever = FALSE;
tp      = FALSE;
imm32   = ZeroExtend(immh:imml:'0', 32);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M Low Overhead Branch Extension*




#### T2: LE variant

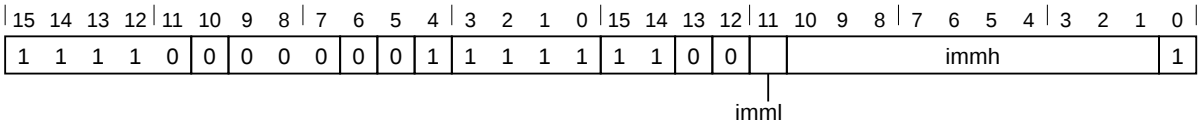
```
LE <label>
```

### Decode for this encoding

```
if !HaveLOBExt() then UNDEFINED;

forever = TRUE;
tp      = FALSE;
imm32   = ZeroExtend(immh:imml:'0', 32);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T3

*Armv8.1-M Low Overhead Branch Extension and MVE*




#### T3: LETP variant

```
LETP LR, <label>
```

### Decode for this encoding

```
if !HaveLOBExt() then UNDEFINED;
if !HaveMve()    then UNDEFINED;
HandleException(CheckCPEnabled(10));

forever = FALSE;
tp      = TRUE;
imm32   = ZeroExtend(immh:imml:'0', 32);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <LR> | LR is used to hold the iteration counter of the loop, this instruction must always use this register. |
| <label> | Specifies the label of the first instruction in the loop body. |

### Operation for all encodings

```
EncodingSpecificOperations();

if tp then
    ExecuteFPCheck();
elsif LTPSIZE != 4 then
    // Tail predicated loop starts should be paired with an LETP loop end.
    // Using a LE instruction in this case is a programming error.
    UFSR.INVSTATE = '1';
    HandleException(CreateException(UsageFault));
if !forever && IsLastLowOverheadLoop() then
    if tp then
        FPSCR.LTPSIZE = 4<2:0>; // Disable loop predication
else
    // Decrement the loop counter
    if !forever then
        LR = LR - (1 << (4 - LTPSIZE))<31:0>;
    // Set up the branch cache info
    jumpAddr = PC - imm32;
    if CCR.LOB == '1' then
        LO_BRANCH_INFO.VALID     = '1';
        LO_BRANCH_INFO.BF        = '0';
        LO_BRANCH_INFO.LF        = if forever then '1' else '0';
        LO_BRANCH_INFO.T16IND    = '0';
        LO_BRANCH_INFO.JUMP_ADDR = jumpAddr<31:1>;
        LO_BRANCH_INFO.END_ADDR  = ThisInstrAddr()<31:1>;
    // Branch to the start of the loop
    BranchTo(jumpAddr);
```

## C2.4.94 LSL (immediate)

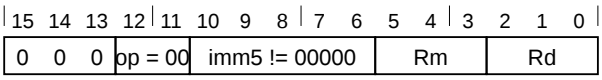
Logical Shift Left (immediate). Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*




**T2 variant**

```
LSL<c>{<q>} {<Rd>,} <Rm>, #<imm>
  // Inside IT block
```

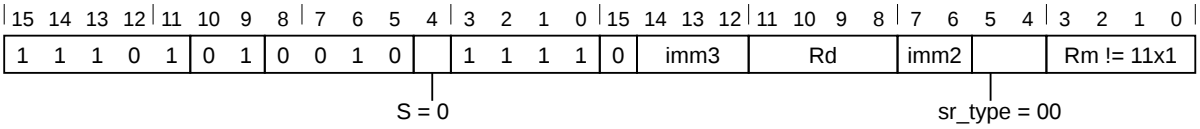
**is equivalent to**

```
MOV<c>{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOV, shift or rotate by value variant**

```
LSL<c>.W {<Rd>,} <Rm>, #<imm>
  // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

**MOV, shift or rotate by value variant**

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.<br>For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.95 LSL (register)

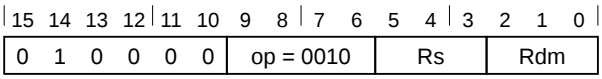
Logical Shift Left (register). Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*




**Logical shift left variant**

```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Inside IT block
```

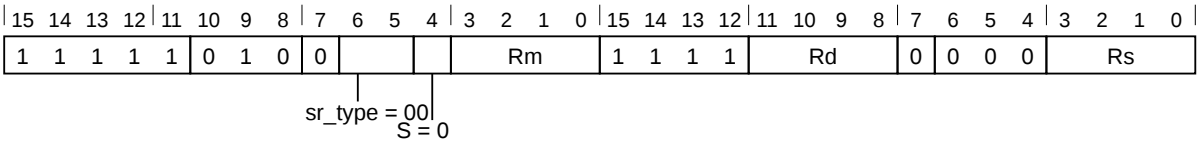
**is equivalent to**

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Non flag setting variant**

```
LSL<c>.W {<Rd>,} <Rm>, <Rs>
  // Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

**Non flag setting variant**

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

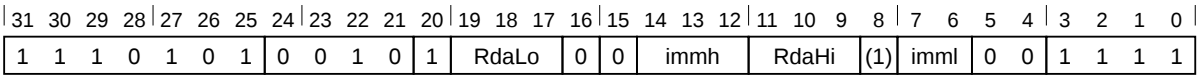
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.96 LSLL (immediate)

Logical Shift Left Long. Logical shift left by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: LSLL variant

```
LSLL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "UQSHL (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

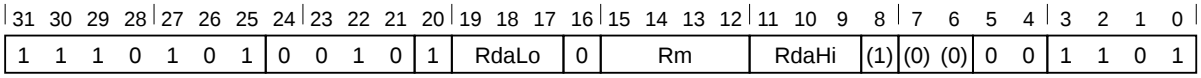
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = UInt(R[dah]:R[dal]);
5      result = (op1 << amount)<63:0>;
6      R[dah] = result<63:32>;
7      R[dal] = result<31:0>;
```

## C2.4.97 LSLL (register)

Logical Shift Left Long. Logical shift left by 0 to 64 bits of a 64 bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: LSLL variant

```
LSLL<c> RdaLo, RdaHi, Rm
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "UQRSHL (register)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  m   = UInt(Rm);
8  if RdaHi == '110' || Rm == '11x1' || Rm == RdaHi:'1' then CONSTRAINED_UNPREDICTABLE;
9  if Rm    == RdaLo:'0' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      amount = SInt(R[m]<7:0>);
5      op1    = UInt(R[dah]:R[dal]);
6      result = (op1 << amount)<63:0>;
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

## C2.4.98 LSLS (immediate)

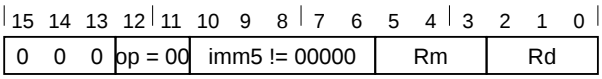
Logical Shift Left, Setting flags (immediate). Logical Shift Left, Setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M*




**T2 variant**

```
LSLS{<q>} {<Rd>,} <Rm>, #<imm>
  // Outside IT block
```

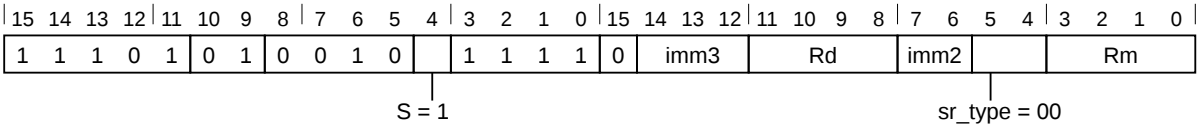
**is equivalent to**

```
MOVS{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOVS, shift or rotate by value variant**

```
LSLS.W {<Rd>,} <Rm>, #<imm>
  // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

**MOVS, shift or rotate by value variant**

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.<br>For encoding T3: is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.99 LSLS (register)

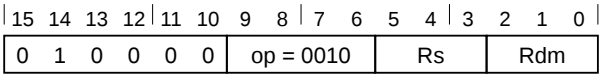
Logical Shift Left, Setting flags (register). Logical Shift Left, Setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M*




**Logical shift left variant**

```
LSLS{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Outside IT block
```

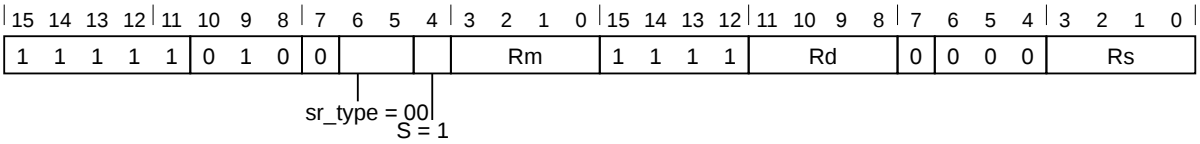
**is equivalent to**

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Flag setting variant**

```
LSLS.W {<Rd>,} <Rm>, <Rs>
  // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

**Flag setting variant**

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.100 LSR (immediate)

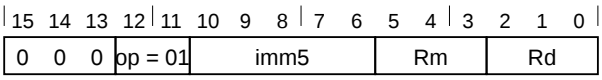
Logical Shift Right (immediate). Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*




**T2 variant**

```
LSR<c>{<q>} {<Rd>,} <Rm>, #<imm>
  // Inside IT block
```

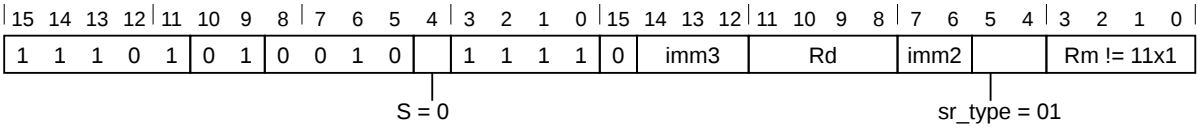
**is equivalent to**

```
MOV<c>{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is the preferred disassembly when `InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOV, shift or rotate by value variant**

```
LSR<c>.W {<Rd>,} <Rm>, #<imm>
  // Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

**MOV, shift or rotate by value variant**

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

| | |
|---|---|
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.101 LSR (register)

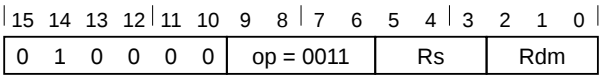
Logical Shift Right (register). Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*




**Logical shift right variant**

```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Inside IT block
```

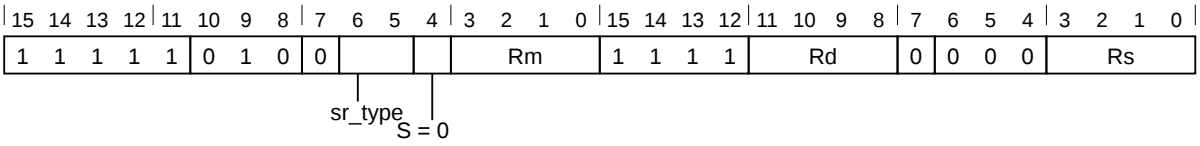
**is equivalent to**

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Non flag setting variant**

```
LSR<c>.W {<Rd>,} <Rm>, <Rs>
  // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

**Non flag setting variant**

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

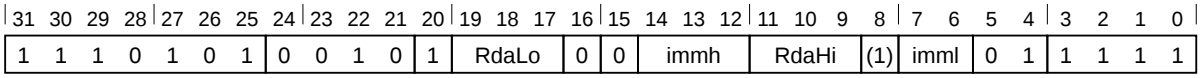
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.102 LSRL (immediate)

Logical Shift Right Long. Logical shift right by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: LSRL variant

```
LSRL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "URSHR (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = UInt(R[dah]:R[dal]);
5      result = (op1 >> amount)<63:0>;
6      R[dah] = result<63:32>;
7      R[dal] = result<31:0>;
```

## C2.4.103 LSRS (immediate)

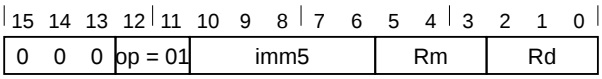
Logical Shift Right, Setting flags (immediate). Logical Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T2

*Armv8-M*




**T2 variant**

```
LSRS{<q>} {<Rd>,} <Rm>, #<imm>
  // Outside IT block
```

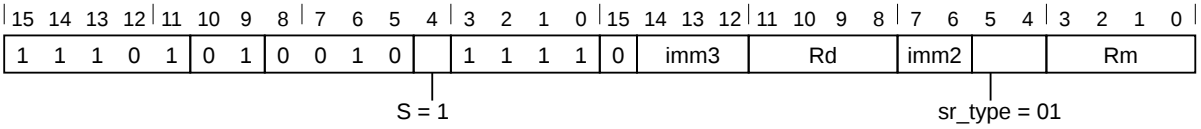
**is equivalent to**

```
MOVS{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T3

*Armv8-M Main Extension only*




**MOVS, shift or rotate by value variant**

```
LSRS.W {<Rd>,} <Rm>, #<imm>
  // Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

**MOVS, shift or rotate by value variant**

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | For encoding T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.<br>For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32. |

## Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.104 LSRS (register)

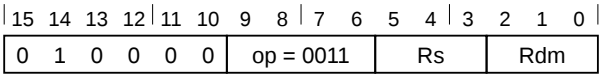
Logical Shift Right, Setting flags (register). Logical Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M*




**Logical shift right variant**

```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Outside IT block
```

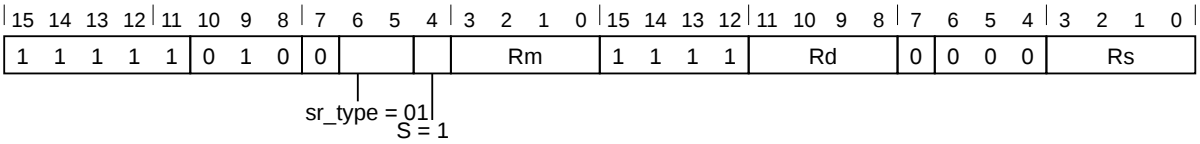
**is equivalent to**

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Flag setting variant**

```
LSRS.W {<Rd>,} <Rm>, <Rs>
  // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

**Flag setting variant**

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

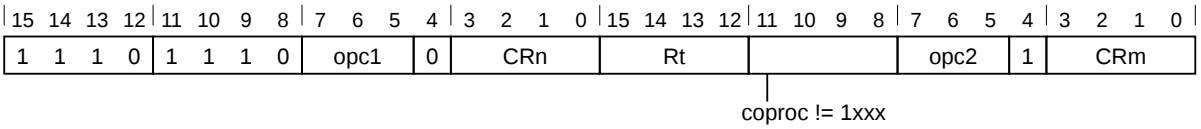
## C2.4.105 MCR, MCR2

Move to Coprocessor from Register. Move to Coprocessor from Register passes the value of a general-purpose register to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

### T1

*Armv8-M Main Extension only*




#### T1 variant

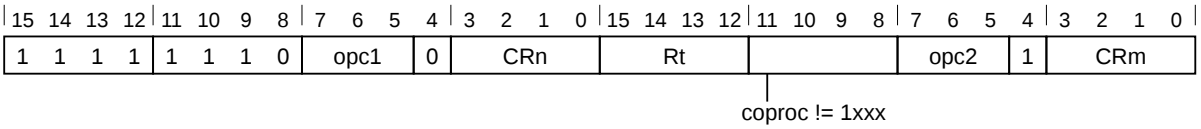
```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  cp = UInt(coproc);
4  if t == 15 || t == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  cp = UInt(coproc);
4  if t == 15 || t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <CRn> | Is the first coprocessor register, encoded in the "CRn" field. |
| <CRm> | Is the second coprocessor register, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendOneWord(R[t], cp, ThisInstr());
```

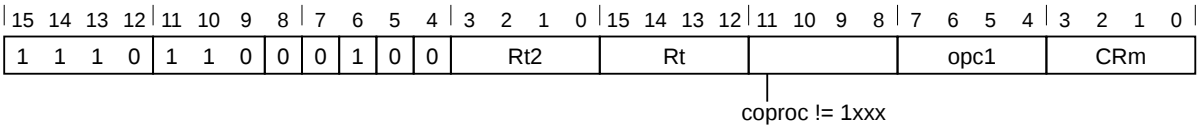
## C2.4.106 MCRR, MCRR2

Move to Coprocessor from two Registers. Move to Coprocessor from two Registers passes the values of two general-purpose registers to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

### T1

*Armv8-M Main Extension only*




#### T1 variant

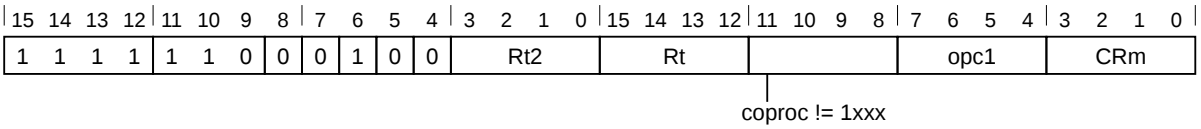
```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
4  if t == 15 || t2 == 15 then UNPREDICTABLE;
5  if t == 13 || t2 == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
4  if t == 15 || t2 == 15 then UNPREDICTABLE;
5  if t == 13 || t2 == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field. |
| <Rt> | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <CRm> | Is a coprocessor register, encoded in the "CRm" field. |

## Operation for all encodings

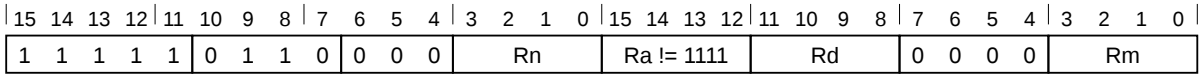
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);
    if !Coproc_Accepted(cp, ThisInstr()) then
        GenerateCoprocessorException();
    else
        Coproc_SendTwoWords(R[t2], R[t], cp, ThisInstr());
```

## C2.4.107 MLA

Multiply Accumulate. Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>`

### Decode for this encoding

```
1  if Ra == '1111' then SEE MUL;
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

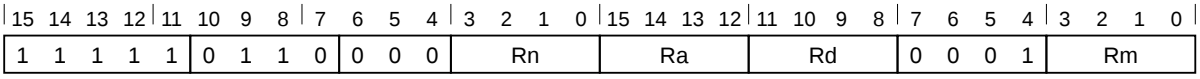
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
4      operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
5      addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
6      result = operand1 * operand2 + addend;
7      R[d] = result<31:0>;
8      if setflags then
9          APSR.N = result<31>;
10         APSR.Z = IsZeroBit(result<31:0>);
11         // APSR.C unchanged
12         // APSR.V unchanged
```

## C2.4.108 MLS

Multiply and Subtract. Multiply and Subtract multiplies two register values, and subtracts the least significant 32 bits of the result from a third register value. These 32 bits do not depend on whether signed or unsigned calculations are performed. The result is written to the destination register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the minuend, encoded in the "Ra" field. |

### Operation for all encodings

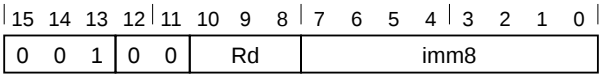
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
4      operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
5      addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
6      result = addend - operand1 * operand2;
7      R[d] = result<31:0>;
```

## C2.4.109 MOV (immediate)

Move (immediate). Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

### T1

*Armv8-M*




#### T1 variant

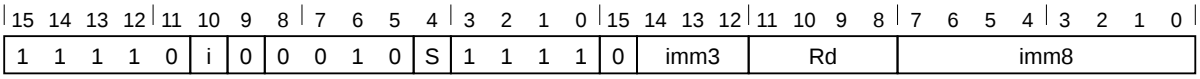
```
MOV<c>{<q>} <Rd>, #<imm8>
  // Inside IT block
MOVS{<q>} <Rd>, #<imm8>
  // Outside IT block
```

### Decode for this encoding

```
1  d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

### T2

*Armv8-M Main Extension only*




#### MOV variant

**Applies when `S == 0`.**

```
MOV<c>.W <Rd>, #<const>
  // Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>
```

#### MOVS variant

**Applies when `S == 1`.**

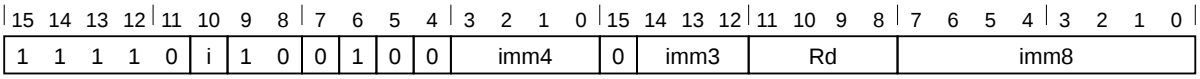
```
MOVS.W <Rd>, #<const>
  // Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
3  if d IN {13,15} then UNPREDICTABLE;
```

### T3

*Armv8-M*




#### T3 variant

```
MOV{<c>}{<q>} <Rd>, #<imm16>
  // <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16>
  // <imm16> can be represented in T1 or T2
```

### Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);  carry = bit UNKNOWN;
if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm8> | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. |
| <imm16> | Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

```
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
```

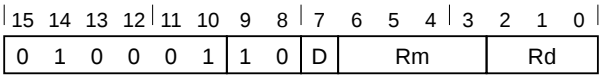
## C2.4.110 MOV (register)

Move (register). Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, RRX.

### T1

*Armv8-M*




#### T1 variant

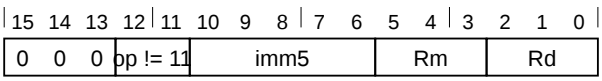
```
MOV{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1 d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
2 (shift_t, shift_n) = (SRType_LSL, 0);
3 if HaveMainExt() then
4     if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### T2

*Armv8-M*




#### T2 variant

```
MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>}
  // Inside IT block
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>}
  // Outside IT block
```

### Decode for this encoding

```
1 if op == '11' then SEE "Related encodings";
2 d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
3 (shift_t, shift_n) = DecodeImmShift(op, imm5);
4 if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

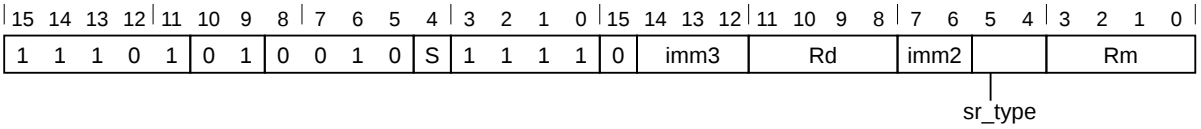
If `op == '00' && imm5 == '00000' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passed its condition code check.
- The instruction executes as NOP, as if it failed its condition code check.
- The instruction executes as `MOV` Rd, Rm.

### T3

*Armv8-M Main Extension only*

- For Armv8.0-M, `C == (0)`.
- For Armv8.1-M, `C == 0`.




**MOV, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

**MOV, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
MOV<c>.W <Rd>, <Rm> {, <shift> #<amount>}
  // Inside IT block
  // and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2
MOV{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

**MOVS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

**MOVS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
MOVS.W <Rd>, <Rm> {, <shift> #<amount>}
  // Outside IT block
  // and <Rd>, <Rm>, <shift>, <amount> can be represented in T1 or T2
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
1  if HasArchVersion(Armv8p1) then
2      if S == '1' && Rm == '11x1' then SEE "Wide shift instructions";
3  if !HaveMainExt() then UNDEFINED;
4  d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
5  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
6  if !setflags && (imm3:imm2:sr_type == '0000000') then
7      if (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
8  else
9      if (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
```

## Alias conditions

| Alias | variant | preferred when |
|---|---|---|
| ASRS (immediate) | T3 | `S == '1' && sr_type == '10'` |
| ASRS (immediate) | T2 | `op == '10' && !InITBlock()` |
| ASR (immediate) | T3 | `S == '0' && sr_type == '10'` |
| ASR (immediate) | T2 | `op == '10' && InITBlock()` |
| LSLS (immediate) | T3 | `S == '1' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '00'` |
| LSLS (immediate) | T2 | `op == '00' && imm5 != '00000' && !InITBlock()` |
| LSL (immediate) | T3 | `S == '0' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '00'` |
| LSL (immediate) | T2 | `op == '00' && imm5 != '00000' && InITBlock()` |
| LSRS (immediate) | T3 | `S == '1' && sr_type == '01'` |
| LSRS (immediate) | T2 | `op == '01' && !InITBlock` |
| LSR (immediate) | T3 | `S =='0' && sr_type == '01'` |
| LSR (immediate) | T2 | `op == '01' && InITBlock()` |
| RORS (immediate) | - | `S == '1' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '11'` |
| ROR (immediate) | - | `S == '0' && imm3:Rd:imm2 != '000xxxx00' && sr_type == '11'` |
| RRXS | - | `S == '1' && imm3 == '000' && imm2 == '00' && sr_type == '11'` |
| RRX | - | `S == '0' && imm3 == '000' && imm2 == '00' && sr_type =='11'` |

## Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> For encoding T1: is the general-purpose destination register, encoded in the "D:Rd" field. If the PC is used:
- The instruction causes a simple 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.

For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding T1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used.
For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.

<shift> For encoding T2: is the type of shift to be applied to the source register, encoded in the "op" field. It can have the following values:

| | |
|---|---|
| LSL | when `sr_type = 00` |
| LSR | when `sr_type = 01` |
| ASR | when `sr_type = 10` |

For encoding T3: is the type of shift to be applied to the source register, encoded in the "sr_type" field. It can have the following values:

| | |
|---|---|
| LSL | when `sr_type = 00` |
| LSR | when `sr_type = 01` |
| ASR | when `sr_type = 10` |
| ROR | when `sr_type = 11` |

<amount> For encoding T2: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm5" field as <amount> modulo 32.
For encoding T3: is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4      if d == 15 then
5          BranchTo(result);  // setflags is always FALSE here
6      else
7          RSPCheck[d] = result;
8          if setflags then
9              APSR.N = result<31>;
```

```
10              APSR.Z = IsZeroBit(result);
11              APSR.C = carry;
12              // APSR.V unchanged
```

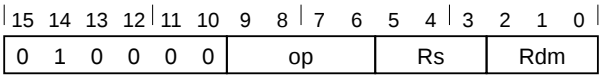
## C2.4.111 MOV, MOVS (register-shifted register)

Move (register-shifted register). Move (register-shifted register) copies a register-shifted register value to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (register), ASR (register), LSLS (register), LSL (register), LSRS (register), LSR (register), RORS (register), ROR (register).

### T1

*Armv8-M*




**Arithmetic shift right variant**

**Applies when `op == 0100`.**

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs>
  // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
  // Outside IT block
```

**Logical shift left variant**

**Applies when `op == 0010`.**

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
  // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs>
  // Outside IT block
```

**Logical shift right variant**

**Applies when `op == 0011`.**

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
  // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
  // Outside IT block
```

**Rotate right variant**

**Applies when `op == 0111`.**

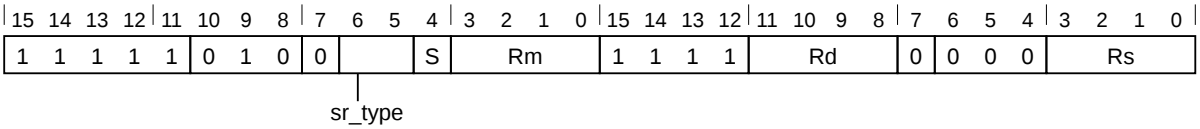
```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
  // Inside IT block
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
  // Outside IT block
```

### Decode for this encoding

```
1  if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
2  d = UInt(Rdm);  m = UInt(Rdm);  s = UInt(Rs);
3  setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);
```

### T2

*Armv8-M Main Extension only*




**Flag setting variant**

**Applies when `S == 1`.**

```
MOVS.W <Rd>, <Rm>, <sr_type> <Rs>
  // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
MOVS{<c>}{<q>} <Rd>, <Rm>, <sr_type> <Rs>
```

**Non flag setting variant**

**Applies when `S == 0`.**

```
MOV<c>.W <Rd>, <Rm>, <sr_type> <Rs>
  // Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
MOV{<c>}{<q>} <Rd>, <Rm>, <sr_type> <Rs>
```

## Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
3 setflags = (S == '1');  shift_t = DecodeRegShift(sr_type);
4 if d IN {13,15} || m IN {13,15} || s IN {13,15} then UNPREDICTABLE;
```

## Alias conditions

| Alias | variant | preferred when |
|---|---|---|
| ASRS (register) | T1 (aritmetic shift right) | `op == '0100' && !InITBlock()` |
| ASRS (register) | T2 (flag setting) | `sr_type == '10' && S == '1'` |
| ASR (register) | T1 (arithmetic shift right) | `op == '0100' && InITBlock()` |
| ASR (register) | T2 (non flag setting) | `sr_type == '10' && S == '0'` |
| LSLS (register) | T1 (logical shift left) | `op == '0010' && !InITBlock()` |
| LSLS (register) | T2 (flag setting) | `sr_type == '00' && S == '1'` |
| LSL (register) | T1 (logical shift left) | `op == '0010' && InITBlock()` |
| LSL (register) | T2 (non flag setting) | `sr_type == '00' && S == '0'` |
| LSRS (register) | T1 (logical shift right) | `op == '0011' && !InITBlock()` |
| LSRS (register) | T2 (flag setting) | `sr_type == '01' && S == '1'` |
| LSR (register) | T1 (logical shift right) | `op == '0011' && InITBlock()` |
| LSR (register) | T2 (non flag setting) | `sr_type =='01' && S == '0'` |
| RORS (register) | T1 (rotate right) | `op == '0111' && !InITBlock()` |
| RORS (register) | T2 (flag setting) | `sr_type == '11' && S =='1'` |
| ROR (register) | T1 (rotate right) | `op == '0111' && InITBlock()` |
| ROR (register) | T2 (non flag setting) | `sr_type == '11' && S == '0'` |

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rdm>` | Is the general-purpose source register and the destination register, encoded in the "Rdm" field. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | Is the general-purpose source register, encoded in the "Rm" field. |
| `<sr_type>` | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>`LSL` when `sr_type = 00` |

LSR when sr_type = 01
ASR when sr_type = 10
ROR when sr_type = 11

<Rs> Is the general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation for all encodings

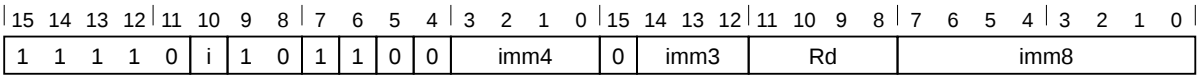
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (result, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

## C2.4.112 MOVT

Move Top. Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

### T1

*Armv8-M*




#### T1 variant

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

### Decode for this encoding

```
1 d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
2 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm16> | Is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field. |

### Operation for all encodings

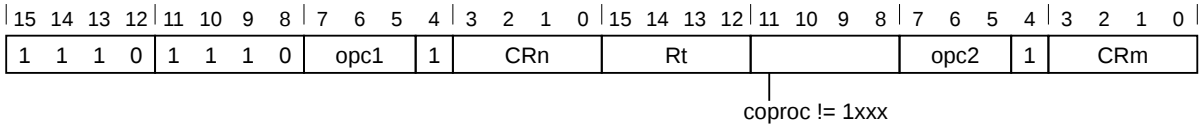
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     R[d]<31:16> = imm16;
4     // R[d]<15:0> unchanged
```

## C2.4.113 MRC, MRC2

Move to Register from Coprocessor. Move to Register from Coprocessor causes a coprocessor to transfer a value to a general-purpose register or to the condition flags.

### T1

*Armv8-M Main Extension only*




#### T1 variant

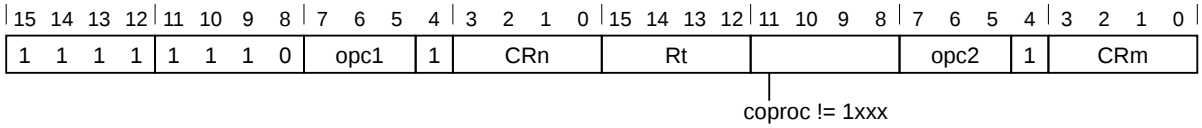
```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

### Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt);  cp = UInt(coproc);
4 if t == 13 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
MRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

### Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt);  cp = UInt(coproc);
4 if t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field. |
| <Rt> | Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the APSR condition flags. |
| <CRn> | Is the first coprocessor register, encoded in the "CRn" field. |
| <CRm> | Is the second coprocessor register, encoded in the "CRm" field. |
| <opc2> | Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);
    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.
```

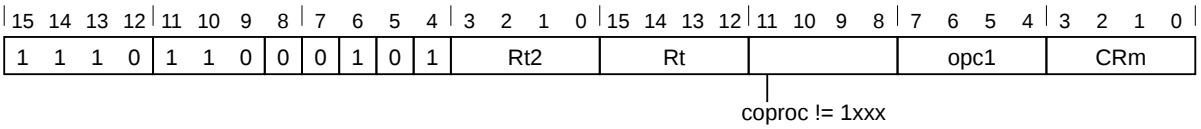
## C2.4.114 MRRC, MRRC2

Move to two Registers from Coprocessor. Move to two Registers from Coprocessor causes a coprocessor to transfer values to two general-purpose registers.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
4  if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
5  if t == 13 || t2 == 13 then UNPREDICTABLE;
```

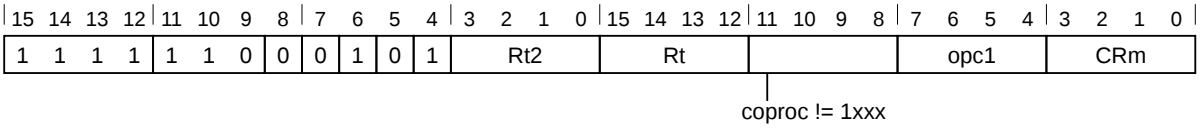
#### CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'} then SEE "Floating-point and MVE";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
4  if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
5  if t == 13 || t2 == 13 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <opc1> | Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field. |
| <Rt> | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rt2> | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| <CRm> | Is a coprocessor register, encoded in the "CRm" field. |

## Operation for all encodings

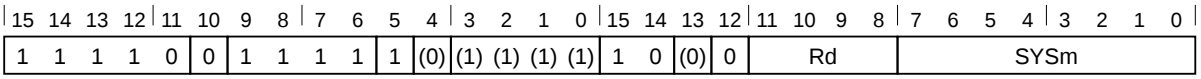
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteCPCheck(cp);
4     if !Coproc_Accepted(cp, ThisInstr()) then
5         GenerateCoprocessorException();
6     else
7         (R[t2], R[t]) = Coproc_GetTwoWords(cp, ThisInstr());
```

## C2.4.115 MRS

Move to Register from Special register. Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose register.

### T1

*Armv8-M*




#### T1 variant

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

### Decode for this encoding

```
1  d = UInt(Rd);
2  if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<spec_reg> Is the special register to be accessed, encoded in the "SYSm" field. It can have the following values:

| | |
|---|---|
| APSR | when SYSm = 00000000 |
| IAPSR | when SYSm = 00000001 |
| EAPSR | when SYSm = 00000010 |
| XPSR | when SYSm = 00000011 |
| IPSR | when SYSm = 00000101 |
| EPSR | when SYSm = 00000110 |
| IEPSR | when SYSm = 00000111 |
| MSP | when SYSm = 00001000 |
| PSP | when SYSm = 00001001 |
| MSPLIM | when SYSm = 00001010 |
| PSPLIM | when SYSm = 00001011 |
| PRIMASK | when SYSm = 00010000 |
| BASEPRI | when SYSm = 00010001 |
| BASEPRI_MAX | when SYSm = 00010010 |
| FAULTMASK | when SYSm = 00010011 |
| CONTROL | when SYSm = 00010100 |
| MSP_NS | when SYSm = 10001000 |
| PSP_NS | when SYSm = 10001001 |
| MSPLIM_NS | when SYSm = 10001010 |
| PSPLIM_NS | when SYSm = 10001011 |
| PRIMASK_NS | when SYSm = 10010000 |
| BASEPRI_NS | when SYSm = 10010001 |
| FAULTMASK_NS | when SYSm = 10010011 |
| CONTROL_NS | when SYSm = 10010100 |
| SP_NS | when SYSm = 10011000 |

The following encodings are UNPREDICTABLE:

- SYSm = 00000100
- SYSm = 000011xx

- SYSm = 00010101
- SYSm = 0001011x
- SYSm = 00011xxx
- SYSm = 001xxxxx
- SYSm = 01xxxxxx
- SYSm = 10000xxx
- SYSm = 100011xx
- SYSm = 10010010
- SYSm = 10010101
- SYSm = 1001011x
- SYSm = 10011001
- SYSm = 1001101x
- SYSm = 100111xx
- SYSm = 101xxxxx
- SYSm = 11xxxxxx

An access to a register not ending in _NS returns the register associated with the current Security state. Access to a register ending in _NS in Secure state returns the Non-secure register. Access to a register ending in _NS in Non-secure state is RAZ/WI. Access to BASEPRI_MAX returns the contents of BASEPRI.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d] = Zeros(32);

    // NOTE: the MSB of SYSm is used to select between either the current
    // domains view of the registers and other domains view of the register.
    // This is required so that the Secure state can access the Non-secure
    // versions of banked registers. For security reasons the Secure versions of
    // the registers are not accessible from the Non-secure state.
    case SYSm<7:3> of
        when '00000'                                /* XPSR accesses */
            if UInt(SYSm) == 4 then UNPREDICTABLE;
            if CurrentModeIsPrivileged() && SYSm<0> == '1' then
                R[d]<8:0> = IPSR.Exception;
            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>;
                if HaveDSPExt() then
                    R[d]<19:16> = APSR<19:16>;
        when '00001'                                /* SP access */
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        R[d] = SP_Main;
                    when '001'
                        R[d] = SP_Process;
                    when '010'
                        if IsSecure() then
                            R[d] = MSPLIM_S.LIMIT:'000';
                        else
                            if HaveMainExt() then
                                R[d] = MSPLIM_NS.LIMIT:'000';
                            else
                                UNPREDICTABLE;
                    when '011'
                        if IsSecure() then
                            R[d] = PSPLIM_S.LIMIT:'000';
                        else
                            if HaveMainExt() then
                                R[d] = PSPLIM_NS.LIMIT:'000';
```

```
                        else
                            UNPREDICTABLE;
                otherwise
                    UNPREDICTABLE;
        when '10001'                            /* SP access - alt domain */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        R[d] = SP_Main_NonSecure;
                    when '001'
                        R[d] = SP_Process_NonSecure;
                    when '010'
                        if HaveMainExt() then
                            R[d] = MSPLIM_NS.LIMIT:'000';
                        else
                            UNPREDICTABLE;
                    when '011'
                        if HaveMainExt() then
                            R[d] = PSPLIM_NS.LIMIT:'000';
                        else
                            UNPREDICTABLE;
                    otherwise
                        UNPREDICTABLE;
        when '00010'                            /* Priority mask or CONTROL access */
            case SYSm<2:0> of
                when '000'
                    if CurrentModeIsPrivileged() then
                        R[d]<0> = PRIMASK.PM;
                when '001'
                    if HaveMainExt() then
                        if CurrentModeIsPrivileged() then
                            R[d]<7:0> = BASEPRI<7:0>;
                    else
                        UNPREDICTABLE;
                when '010'
                    if HaveMainExt() then
                        if CurrentModeIsPrivileged() then
                            R[d]<7:0> = BASEPRI<7:0>;
                    else
                        UNPREDICTABLE;
                when '011'
                    if HaveMainExt() then
                        if CurrentModeIsPrivileged() then
                            R[d]<0> = FAULTMASK.FM;
                    else
                        UNPREDICTABLE;
                when '100'
                    if HaveMveOrFPExt() && IsSecure() then
                        R[d]<3:0> = CONTROL<3:0>;
                    elsif HaveMveOrFPExt() then
                        R[d]<2:0> = CONTROL<2:0>;
                    else
                        R[d]<1:0> = CONTROL<1:0>;
                otherwise
                    UNPREDICTABLE;
        when '10010'                            /* Priority mask or CONTROL access - alt
            domain */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        if CurrentModeIsPrivileged() then
                            R[d]<0> = PRIMASK_NS.PM;
                    when '001'
                        if HaveMainExt() then
                            if CurrentModeIsPrivileged() then
                                R[d]<7:0> = BASEPRI_NS<7:0>;
                        else
```

```
                        UNPREDICTABLE;
                when '011'
                    if HaveMainExt() then
                        if CurrentModeIsPrivileged() then
                            R[d]<0> = FAULTMASK_NS.FM;
                    else
                        UNPREDICTABLE;
                when '100'
                    if HaveMveOrFPExt() then
                        R[d]<2:0> = CONTROL_NS<2:0>;
                    else
                        R[d]<1:0> = CONTROL_NS<1:0>;
                otherwise
                    UNPREDICTABLE;
        when '10011'                            /* SP_NS - Non-secure stack pointer */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        R[d] = _SP(LookUpSP_with_security_mode(FALSE, CurrentMode()));
                    otherwise
                        UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `SYSm not valid special register`, then one of the following behaviors must occur:

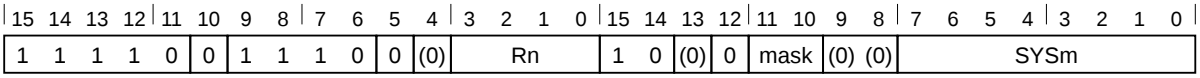
- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

## C2.4.116 MSR (register)

Move to Special register from Register. Move to Special register from Register moves the value of a general-purpose register to the specified special-purpose register.

### T1

*Armv8-M*




#### T1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

### Decode for this encoding

```
1  n = UInt(Rn);
2  if HaveMainExt() then
3      if mask == '00' || (mask != '10' && !(UInt(SYSm) IN {0..3})) then UNPREDICTABLE;
4  else
5      if mask != '10' then UNPREDICTABLE;
6  if n IN {13,15} then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `combination of SYSm and mask not supported`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction treats mask and SYSm as UNKNOWN.

### Encoding conditions

| _<bits> | Effect | mask encoding | Notes |
|---|---|---|---|
| _nzcvq | Write the N, Z, C, V, Q bits, APSR[31:27] | 0b10 | Always supported |
| _g | Write the GE[3:0] bits, APSR[19:16] | 0b01 | Supported only if the PE implements the DSP extension. |
| _nzcvqg | Write the N, Z, C, V, Q, GE[3:0] bits | 0b11 | |

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<spec_reg> Is the special register to be accessed, encoded in the "SYSm" field. It can have the following values:

| | |
|---|---|
| APSR | when SYSm = 00000000 |
| IAPSR | when SYSm = 00000001 |
| EAPSR | when SYSm = 00000010 |
| XPSR | when SYSm = 00000011 |
| IPSR | when SYSm = 00000101 |
| EPSR | when SYSm = 00000110 |
| IEPSR | when SYSm = 00000111 |
| MSP | when SYSm = 00001000 |
| PSP | when SYSm = 00001001 |
| MSPLIM | when SYSm = 00001010 |
| PSPLIM | when SYSm = 00001011 |

| | |
|---|---|
| PRIMASK | when SYSm = 00010000 |
| BASEPRI | when SYSm = 00010001 |
| BASEPRI_MAX | when SYSm = 00010010 |
| FAULTMASK | when SYSm = 00010011 |
| CONTROL | when SYSm = 00010100 |
| MSP_NS | when SYSm = 10001000 |
| PSP_NS | when SYSm = 10001001 |
| MSPLIM_NS | when SYSm = 10001010 |
| PSPLIM_NS | when SYSm = 10001011 |
| PRIMASK_NS | when SYSm = 10010000 |
| BASEPRI_NS | when SYSm = 10010001 |
| FAULTMASK_NS | when SYSm = 10010011 |
| CONTROL_NS | when SYSm = 10010100 |
| SP_NS | when SYSm = 10011000 |

The following encodings are UNPREDICTABLE:

- SYSm = 00000100
- SYSm = 000011xx
- SYSm = 00010101
- SYSm = 0001011x
- SYSm = 00011xxx
- SYSm = 001xxxxx
- SYSm = 01xxxxxx
- SYSm = 10000xxx
- SYSm = 100011xx
- SYSm = 10010010
- SYSm = 10010101
- SYSm = 1001011x
- SYSm = 10011001
- SYSm = 1001101x
- SYSm = 100111xx
- SYSm = 101xxxxx
- SYSm = 11xxxxxx

An access to a register not ending in _NS returns the register associated with the current Security state. Access to a register ending in _NS in Secure state returns the Non-secure register. Access to a register ending in _NS in Non-secure state is RAZ/WI. Access to BASEPRI_MAX writes to BASEPRI if the priority that is written is higher than the existing priority in BASEPRI. Otherwise, the access is ignored.

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    // NOTE: the MSB of SYSm is used to select between either the current
    // domains view of the registers and other domains view of the register.
    // This is required to that the Secure state can access the Non-secure
    // versions of banked registers. For security reasons the Secure versions of
    // the registers are not accessible from the Non-secure state.
    case SYSm<7:3> of
        when '00000'                            /* XPSR accesses */
            if UInt(SYSm) == 4 then UNPREDICTABLE;
            if SYSm<2> == '0' then              /* Include APSR  */
                if mask<0> == '1' then          /* GE[3:0] bits  */
                    if !HaveDSPExt() then
                        UNPREDICTABLE;
                    else
                        APSR<19:16> = R[n]<19:16>;
                if mask<1> == '1' then          /* N, Z, C, V, Q bits */
                    APSR<31:27> = R[n]<31:27>;
```

```
        when '00001'                            /* SP access */
            if CurrentModeIsPrivileged() then
                case SYSm<2:0> of
                    when '000'
                        // MSR not subject to SP limit, write directly to register.
                        if IsSecure() then
                            exc = _SP(RNamesSP_Main_Secure, FALSE, TRUE, R[n]<31:2>:'00');
                        else
                            exc = _SP(RNamesSP_Main_NonSecure, FALSE, TRUE, R[n]<31:2>:'00');
                        assert exc.fault == NoFault;
                    when '001'
                        // MSR not subject to SP limit, write directly to register.
                        if IsSecure() then
                            exc = _SP(RNamesSP_Process_Secure, FALSE, TRUE, R[n]<31:2>:'00');
                        else
                            exc = _SP(RNamesSP_Process_NonSecure, FALSE, TRUE, R[n]<31:2>:'00
                                ');
                        assert exc.fault == NoFault;
                    when '010'
                        if IsSecure() then
                            MSPLIM_S.LIMIT = R[n]<31:3>;
                        else
                            if HaveMainExt() then
                                MSPLIM_NS.LIMIT = R[n]<31:3>;
                            else
                                UNPREDICTABLE;
                    when '011'
                        if IsSecure() then
                            PSPLIM_S.LIMIT = R[n]<31:3>;
                        else
                            if HaveMainExt() then
                                PSPLIM_NS.LIMIT = R[n]<31:3>;
                            else
                                UNPREDICTABLE;
                    otherwise
                        UNPREDICTABLE;
        when '10001'                            /* SP access - alt domain */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        // MSR not subject to SP limit, write directly to register.
                        exc = _SP(RNamesSP_Main_NonSecure, FALSE, TRUE, R[n]<31:2>:'00');
                        assert exc.fault == NoFault;
                    when '001'
                        // MSR not subject to SP limit, write directly to register.
                        exc = _SP(RNamesSP_Process_NonSecure, FALSE, TRUE, R[n]<31:2>:'00');
                        assert exc.fault == NoFault;
                    when '010'
                        if HaveMainExt() then
                            MSPLIM_NS.LIMIT = R[n]<31:3>;
                        else
                            UNPREDICTABLE;
                    when '011'
                        if HaveMainExt() then
                            PSPLIM_NS.LIMIT = R[n]<31:3>;
                        else
                            UNPREDICTABLE;
                    otherwise
                        UNPREDICTABLE;
        when '00010'                            /* Priority mask or CONTROL access */
            case SYSm<2:0> of
                when '000'
                    if CurrentModeIsPrivileged() then
                        PRIMASK.PM = R[n]<0>;
                when '001'
                    if CurrentModeIsPrivileged() then
                        if HaveMainExt() then
                            BASEPRI<7:0> = R[n]<7:0>;
```

```
                        else
                            UNPREDICTABLE;
                when '010'
                    if CurrentModeIsPrivileged() then
                        if HaveMainExt() then
                            if (R[n]<7:0> != '00000000') &&
                               (UInt(R[n]<7:0>) < UInt(BASEPRI<7:0>) || BASEPRI<7:0> == '
                                   00000000') then
                                BASEPRI<7:0> = R[n]<7:0>;
                        else
                            UNPREDICTABLE;
                when '011'
                    if CurrentModeIsPrivileged() then
                        if HaveMainExt() then
                            if ExecutionPriority() > -1 || R[n]<0> == '0' then
                                FAULTMASK.FM = R[n]<0>;
                        else
                            UNPREDICTABLE;
                when '100'
                    if CurrentModeIsPrivileged() then
                        CONTROL.nPRIV = R[n]<0>;
                        CONTROL.SPSEL = R[n]<1>;
                        if HaveMveOrFPExt() && (IsSecure() || NSACR.CP10 == '1') then
                            CONTROL.FPCA = R[n]<2>;
                        if HaveMveOrFPExt() && IsSecure() then
                            CONTROL_S.SFPA = R[n]<3>;
                otherwise
                    UNPREDICTABLE;
        when '10010'                            /* Priority mask or CONTROL access - alt
            domain */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentModeIsPrivileged() && CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        PRIMASK_NS.PM = R[n]<0>;
                    when '001'
                        if HaveMainExt() then
                            BASEPRI_NS<7:0> = R[n]<7:0>;
                        else
                            UNPREDICTABLE;
                    when '011'
                        if HaveMainExt() then
                            if ExecutionPriority() > -1 || R[n]<0> == '0' then
                                FAULTMASK_NS.FM = R[n]<0>;
                        else
                            UNPREDICTABLE;
                    when '100'
                        CONTROL_NS.nPRIV = R[n]<0>;
                        CONTROL_NS.SPSEL = R[n]<1>;
                        if HaveMveOrFPExt() then CONTROL_NS.FPCA = R[n]<2>;
                    otherwise
                        UNPREDICTABLE;
        when '10011'                            /* SP_NS - Non-secure stack pointer */
            if !HaveSecurityExt() then UNPREDICTABLE;
            if CurrentState == SecurityState_Secure then
                case SYSm<2:0> of
                    when '000'
                        spName = LookUpSP_with_security_mode(FALSE, CurrentMode());
                        // MSR SP_NS is subject to SP limit check.
                        - = _SP(spName, FALSE, FALSE, R[n]);
                    otherwise
                        UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `SYSm not valid special register`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction treats SYSm as UNKNOWN.

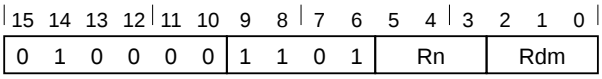
## C2.4.117 MUL

Multiply. Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

It can optionally update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

### T1

*Armv8-M*




#### T1 variant

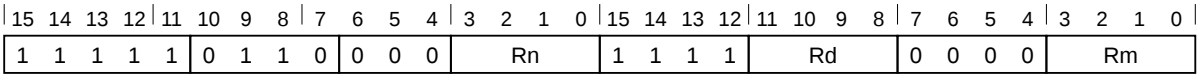
```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>}
  // Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>}
  // Outside IT block
```

### Decode for this encoding

```
1  d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
MUL<c>.W <Rd>, <Rn>{, <Rm>}
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. If omitted, <Rd> is used. |

## Operation for all encodings

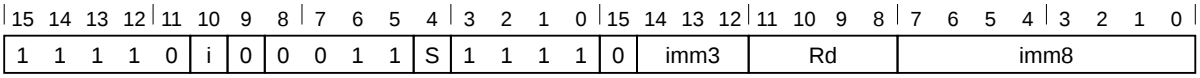
```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result<31:0>);
        // APSR.C unchanged
        // APSR.V unchanged
```

## C2.4.118 MVN (immediate)

Bitwise NOT (immediate). Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value.

### T1

*Armv8-M Main Extension only*




**MVN variant**

**Applies when `S == 0`.**

```
MVN{<c>}{<q>} <Rd>, #<const>
```

**MVNS variant**

**Applies when `S == 1`.**

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd);  setflags = (S == '1');
3 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4 if d IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

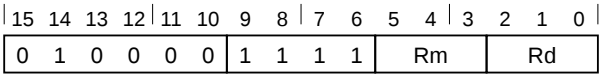
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     result = NOT(imm32);
4     R[d] = result;
5     if setflags then
6         APSR.N = result<31>;
7         APSR.Z = IsZeroBit(result);
8         APSR.C = carry;
9         // APSR.V unchanged
```

## C2.4.119 MVN (register)

Bitwise NOT (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.

### T1

*Armv8-M*




#### T1 variant

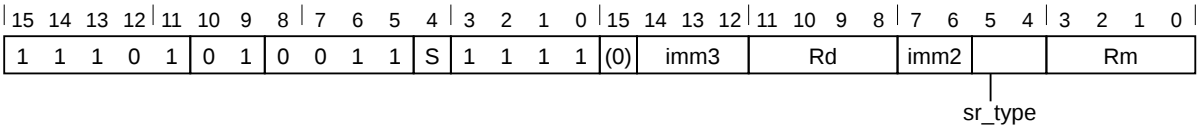
```
MVN<c>{<q>} <Rd>, <Rm>
  // Inside IT block
MVNS{<q>} <Rd>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### MVN, rotate right with extend variant

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### MVN, shift or rotate by value variant

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
MVN<c>.W <Rd>, <Rm>
  // Inside IT block, and <Rd>, <Rm> can be represented in T1
MVN{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

#### MVNS, rotate right with extend variant

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

#### MVNS, shift or rotate by value variant

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
MVNS.W <Rd>, <Rm>
  // Outside IT block, and <Rd>, <Rm> can be represented in T1
MVNS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

### Operation for all encodings

```
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
```

## C2.4.120 NOP

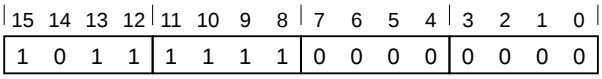
No Operation. No Operation does nothing.

The architecture makes no guarantees about any timing effects of including a NOP instruction.

This is a NOP-compatible hint.

### T1

*Armv8-M*




#### T1 variant

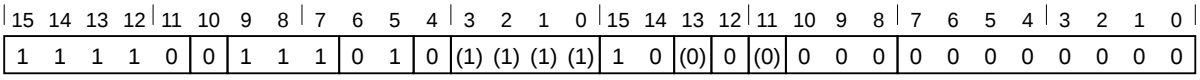
```
NOP{<c>}{<q>}
```

### Decode for this encoding

```
// No additional decoding required
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
NOP{<c>}.W
```

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

### Operation for all encodings

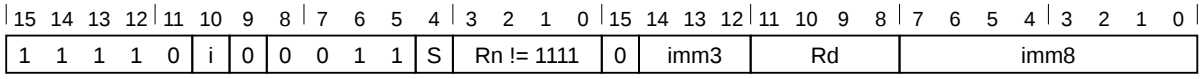
```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

## C2.4.121 ORN (immediate)

Logical OR NOT (immediate). Logical OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**Flag setting variant**

**Applies when `S == 1`.**

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**Non flag setting variant**

**Applies when `S == 0`.**

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE "MVN (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

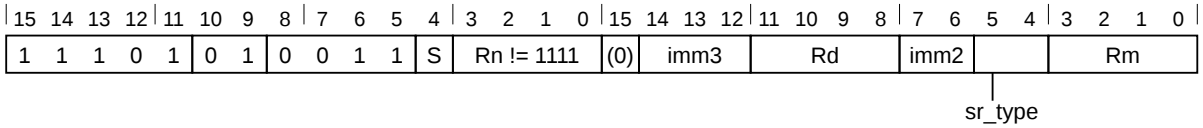
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     result = R[n] OR NOT(imm32);
4     R[d] = result;
5     if setflags then
6         APSR.N = result<31>;
7         APSR.Z = IsZeroBit(result);
8         APSR.C = carry;
9         // APSR.V unchanged
```

## C2.4.122 ORN (register)

Logical OR NOT (register). Logical OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**ORN, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ORN, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**ORNS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ORNS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE "MVN (register)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
4 (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>`LSL` when `sr_type = 00`<br>`LSR` when `sr_type = 01`<br>`ASR` when `sr_type = 10`<br>`ROR` when `sr_type = 11` |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

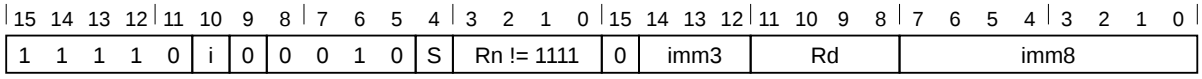
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

## C2.4.123 ORR (immediate)

Logical OR (immediate). Logical OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**ORR variant**

**Applies when `S == 0`.**

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**ORRS variant**

**Applies when `S == 1`.**

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE "MOV (immediate)";
2 if !HaveMainExt() then UNDEFINED;
3 d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
4 (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
5 if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

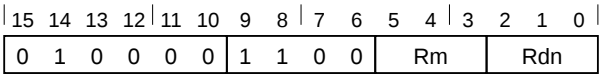
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     result = R[n] OR imm32;
4     R[d] = result;
5     if setflags then
6         APSR.N = result<31>;
7         APSR.Z = IsZeroBit(result);
8         APSR.C = carry;
9         // APSR.V unchanged
```

## C2.4.124 ORR (register)

Logical OR (register). Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
ORRS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

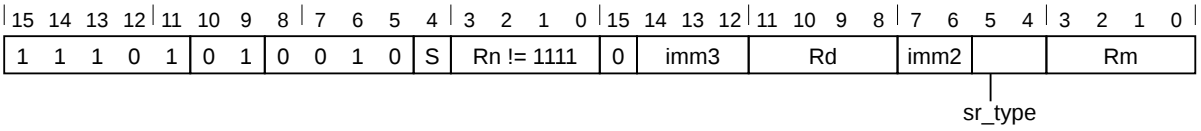
### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*

- For Armv8.0-M, `C == (0)`.
- For Armv8.1-M, `C == 0`.




**ORR, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ORR, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ORR<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**ORRS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**ORRS, shift or rotate by value variant**

**Applies when `S == 1&& !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
ORRS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if HasArchVersion(Armv8p1) then
    if S == '1' && Rm == '11x1' then SEE "Wide shift instructions";
if Rn == '1111' then SEE "MOV (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

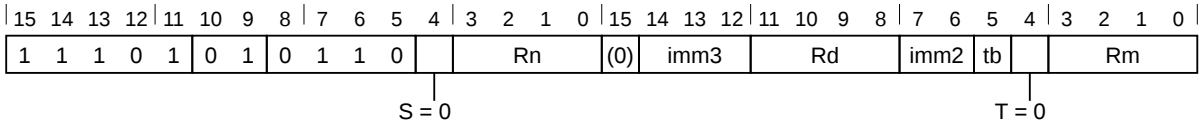
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

## C2.4.125 PKHBT, PKHTB

Pack Halfword. Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

### T1

*Armv8-M DSP Extension only*




**PKHBT variant**

**Applies when `tb == 0`.**

```
PKHBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, LSL #<imm>}
  // tbform == FALSE
```

**PKHTB variant**

**Applies when `tb == 1`.**

```
PKHTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ASR #<imm>}
  // tbform == TRUE
```

### Decode for this encoding

```
1 if S == '1' || T == '1' then UNDEFINED;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
4 (shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
5 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<Rc> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rq> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<imm> The shift to apply to the value read from <Rm>, encoded in imm3:imm2. For PKHBT, it is one of:

1-31 Left shift by specified number of bits, encoded as a binary number.

omitted No shift, encoded as 0b00000.

<imm> The shift to apply to the value read from <Rm>, encoded in imm3:imm2. For PKHTB, it is one of:

1-32 Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

omitted Instruction is a pseudo-instruction and is assembled as though PKHBT<c><q> <Rd>, <Rm>, <Rn> had been written.

For both variants an assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

### Operation for all encodings

```
1 if ConditionPassed() then
```

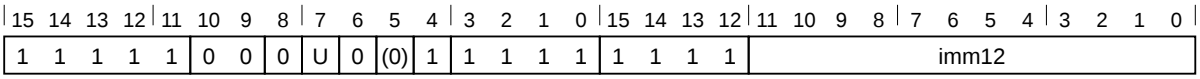
```
    EncodingSpecificOperations();
    operand2 = Shift(R[m], shift_t, shift_n, APSR.C);  // APSR.C ignored
    bits(32) result;
    result<15:0>  = if tbform then operand2<15:0> else R[n]<15:0>;
    result<31:16> = if tbform then R[n]<31:16>    else operand2<31:16>;
    R[d] = result;
```

## C2.4.126 PLD (literal)

Preload Data (literal). Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
PLD{<c>}{<q>} <label>
  // Preferred syntax
PLD{<c>}{<q>} [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <label> | The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
4      Hint_PreloadData(address);
```

## C2.4.127 PLD, PLDW (immediate)

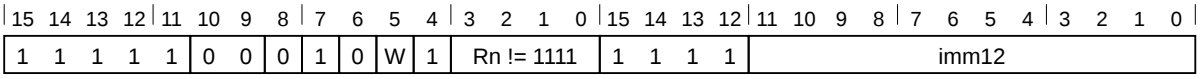
Preload Data (immediate). Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` is IMPLEMENTATION DEFINED.

### T1

*Armv8-M Main Extension only*




**Preload read variant**

**Applies when `W == 0`.**

```
PLD{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

**Preload write variant**

**Applies when `W == 1`.**

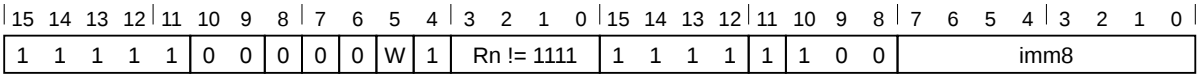
```
PLDW{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "PLD (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

### T2

*Armv8-M Main Extension only*




**Preload read variant**

**Applies when `W == 0`.**

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

**Preload write variant**

**Applies when `W == 1`.**

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "PLD (literal)";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see C2.4.126 *PLD (literal)* on page 713. |
| + | Specifies the offset is added to the base register. |
| <imm> | For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     address = if add then (R[n] + imm32) else (R[n] - imm32);
4     if is_pldw then
5         Hint_PreloadDataForWrite(address);
6     else
7         Hint_PreloadData(address);
```

## C2.4.128 PLD, PLDW (register)

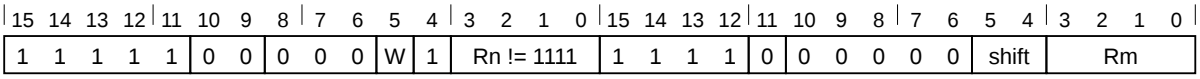
Preload Data (register). Preload Data is a memory hint instruction that can signal the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

The `PLD` instruction signals that the likely memory access is a read, and the `PLDW` instruction signals that it is a write.

The effect of a `PLD` or `PLDW` is IMPLEMENTATION DEFINED.

### T1

*Armv8-M Main Extension only*




**Preload read variant**

**Applies when `W == 0`.**

```
PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

**Preload write variant**

**Applies when `W == 1`.**

```
PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE "PLD (literal)";
2 if !HaveMainExt() then UNDEFINED;
3 n = UInt(Rn);  m = UInt(Rm);  add = TRUE;  is_pldw = (W == '1');
4 (shift_t, shift_n) = (SRType_LSL, UInt(shift));
5 if m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `+` | Specifies the index register that is added to the base register. |
| `<Rm>` | Is the general purpose index register, encoded in the "Rm" field. |
| `<amount>` | Is the shift amount, in the range 0-3, defaulting to 0 and encoded in the "imm2" field. |

### Operation for all encodings

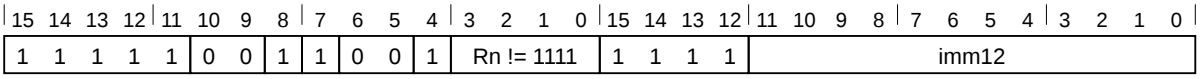
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     offset = Shift(R[m], shift_t, shift_n, APSR.C);
4     address = if add then (R[n] + offset) else (R[n] - offset);
5     if is_pldw then
6         Hint_PreloadDataForWrite(address);
7     else
8         Hint_PreloadData(address);
```

## C2.4.129 PLI (immediate, literal)

Preload Instruction (immediate, literal). Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

### T1

*Armv8-M Main Extension only*




#### T1 variant

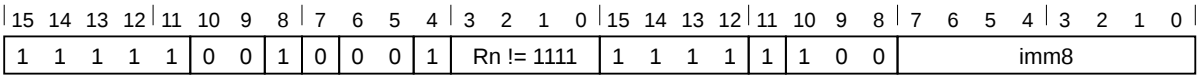
```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "encoding T3";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

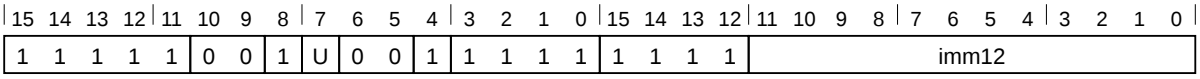
```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "encoding T3";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

```
PLI{<c>}{<q>} <label>
  // Preferred syntax
PLI{<c>}{<q>} [PC, #{+/-}<imm>]
  // Alternative syntax
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <label> | The label of the instruction that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. The offset must be in the range -4095 to 4095. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | For encoding T1: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T2: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.<br>For encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field. |

### Operation for all encodings

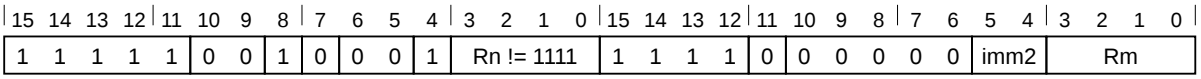
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     base = if n == 15 then Align(PC,4) else R[n];
4     address = if add then (base + imm32) else (base - imm32);
5     Hint_PreloadInstr(address);
```

## C2.4.130 PLI (register)

Preload Instruction (register). Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "PLI (immediate, literal)";
2  if !HaveMainExt() then UNDEFINED;
3  n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
4  (shift_t, shift_n) = (SRType_LSL, UInt(imm2));
5  if m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <amount> | Is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      offset = Shift(R[m], shift_t, shift_n, APSR.C);
4      address = if add then (R[n] + offset) else (R[n] - offset);
5      Hint_PreloadInstr(address);
```

## C2.4.131 POP (multiple registers)

Pop multiple registers from stack. Pop multiple registers from stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point above the loaded data.

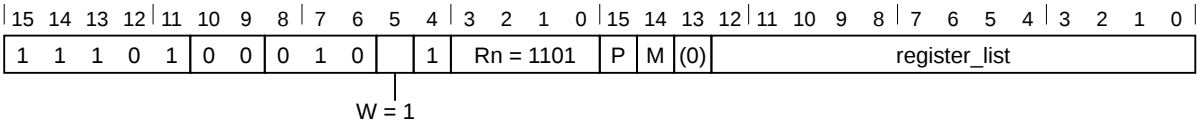
If the registers loaded include the PC, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

This instruction is an alias of the LDM, LDMIA, LDMFD instruction. This means that:

- The encodings in this description are named to match the encodings of LDM, LDMIA, LDMFD.
- The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*




**T2 variant**

```
POP{<c>}{<q>} <registers>
```

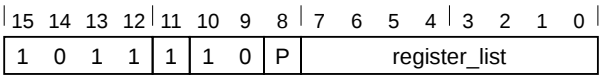
**is equivalent to**

```
LDM{<c>}{<q>} SP!, <registers>
```

**and is the preferred disassembly when `BitCount(register_list) > 1`.**

### T3

*Armv8-M*




**T3 variant**

```
POP{<c>}{<q>} <registers>
```

**is equivalent to**

```
LDM{<c>}{<q>} SP!, <registers>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

`<registers>` For encoding T2: is a list of two or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain one of the LR or the PC. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. If the PC is in the list, the "P" field is set to 1, otherwise it defaults to 0. The PC can be in the list. If it is, the instruction branches to the address loaded to the PC, and: If the PC is in the list:

- The LR must not be in the list.
- The instruction must be either outside any IT block, or the last instruction in an IT block.

For encoding T3: is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the PC. If the PC is in the list, the "P" field is set to 1, otherwise this field defaults to 0. If the PC is in the list, the instruction must be either outside any IT block, or the last instruction in an IT block.

## Operation for all encodings

The description of LDM, LDMIA, LDMFD gives the operational pseudocode for this instruction.

## C2.4.132 POP (single register)

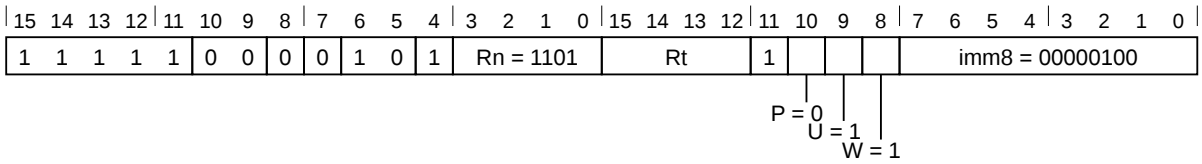
Pop single register from stack. Pop single register from stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point above the loaded data.

This instruction is an alias of the LDR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of LDR (immediate).
- The description of LDR (immediate) gives the operational pseudocode for this instruction.

### T4

*Armv8-M Main Extension only*




**Post-indexed variant**

```
POP{<c>}{<q>} <register>
```

**is equivalent to**

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<register>` | Is the general-purpose register `<Rt>` to be loaded surrounded by { and }. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, provided the instruction is either outside an IT block or the last instruction of an IT block. If the PC is used, the instruction branches to the address (data) loaded to the PC. |

### Operation for all encodings

The description of LDR (immediate) gives the operational pseudocode for this instruction.

## C2.4.133 PSSBB

Physical Speculative Store Bypass Barrier. Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the PSSBB.

### T1

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

#### T1 variant

`PSSBB{<q>}`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      SpeculativeSynchronizationBarrier();
```

## C2.4.134 PUSH (multiple registers)

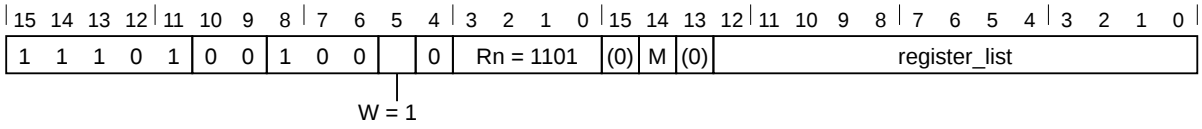
Push multiple registers to stack. Push multiple registers to stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending below the address in SP, and updates SP to point to the start of the stored data.

This instruction is an alias of the STMDB, STMFD instruction. This means that:

- The encodings in this description are named to match the encodings of STMDB, STMFD.
- The description of STMDB, STMFD gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*




**T1 variant**

```
PUSH{<c>}{<q>} <registers>
```

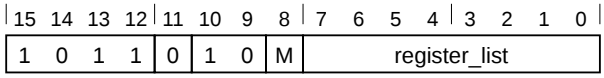
**is equivalent to**

```
STMDB{<c>}{<q>} SP!, <registers>
```

**and is the preferred disassembly when `BitCount(register_list) > 1`.**

### T2

*Armv8-M*




**T2 variant**

```
PUSH{<c>}{<q>} <registers>
```

**is equivalent to**

```
STMDB{<c>}{<q>} SP!, <registers>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <registers> | For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.<br>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0. |

### Operation for all encodings

The description of STMDB, STMFD gives the operational pseudocode for this instruction.

## C2.4.135 PUSH (single register)

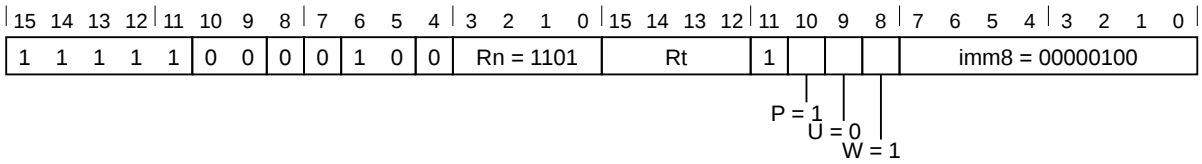
Push single register to stack. Push single register to stack stores a single general-purpose register to the stack, storing to the 32-bit word below the address in SP, and updates SP to point to the start of the stored data.

This instruction is an alias of the STR (immediate) instruction. This means that:

- The encodings in this description are named to match the encodings of STR (immediate).
- The description of STR (immediate) gives the operational pseudocode for this instruction.

### T4

*Armv8-M Main Extension only*




**Pre-indexed variant**

```
PUSH{<c>}{<q>} <register>
  // Standard syntax
```

**is equivalent to**

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<register>` | Is the general-purpose register `<Rt>` to be stored surrounded by { and }. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |

### Operation for all encodings

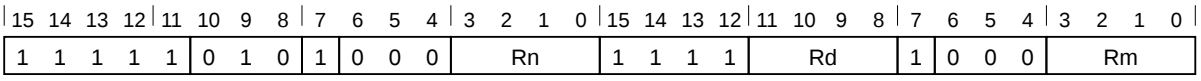
The description of STR (immediate) gives the operational pseudocode for this instruction.

## C2.4.136 QADD

Saturating Add. Saturating Add adds two register values, saturates the result to the 32-bit signed integer range $-2^{31}$ to $2^{31}-1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

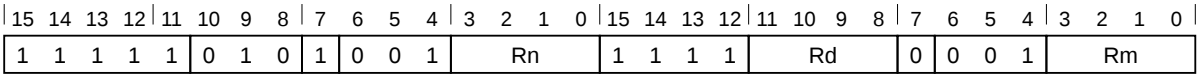
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
4      if sat then
5          APSR.Q = '1';
```

## C2.4.137 QADD16

Saturating Add 16. Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range $-2^{15}$ to $2^{15}-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

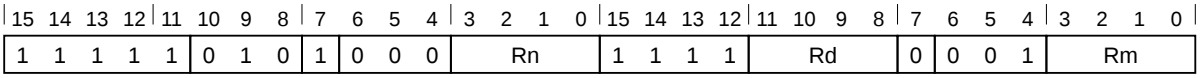
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
4      sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
5      bits(32) result;
6      result<15:0>  = SignedSat(sum1, 16);
7      result<31:16> = SignedSat(sum2, 16);
8      R[d] = result;
```

## C2.4.138 QADD8

Saturating Add 8. Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range $-2^7$ to $2^7-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

`QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>`

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

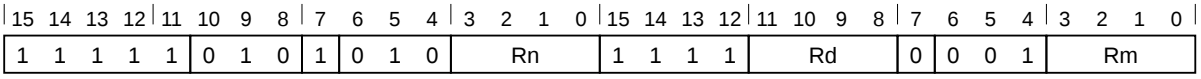
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
4      sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
5      sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
6      sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
7      bits(32) result;
8      result<7:0>   = SignedSat(sum1, 8);
9      result<15:8>  = SignedSat(sum2, 8);
10     result<23:16> = SignedSat(sum3, 8);
11     result<31:24> = SignedSat(sum4, 8);
12     R[d] = result;
```

## C2.4.139 QASX

Saturating Add and Subtract with Exchange. Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range $-2^{15}$ to $2^{15}-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

`QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>`

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

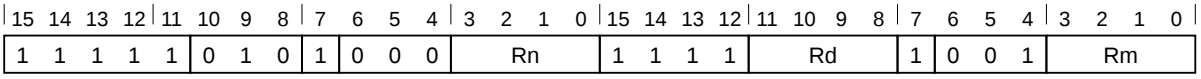
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
4      sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
5      bits(32) result;
6      result<15:0>  = SignedSat(diff, 16);
7      result<31:16> = SignedSat(sum,  16);
8      R[d] = result;
```

## C2.4.140 QDADD

Saturating Double and Add. Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31}$ to $2^{31}-1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

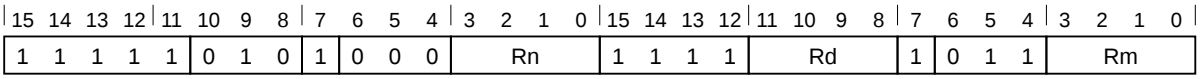
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
4      (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
5      if sat1 || sat2 then
6          APSR.Q = '1';
```

## C2.4.141 QDSUB

Saturating Double and Subtract. Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31}$ to $2^{31}-1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

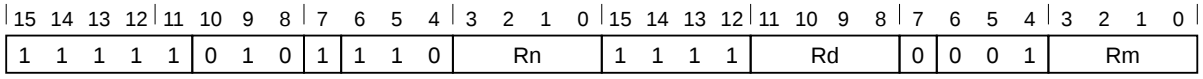
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
4      (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
5      if sat1 || sat2 then
6          APSR.Q = '1';
```

## C2.4.142 QSAX

Saturating Subtract and Add with Exchange. Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range $-2^{15}$ to $2^{15}-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

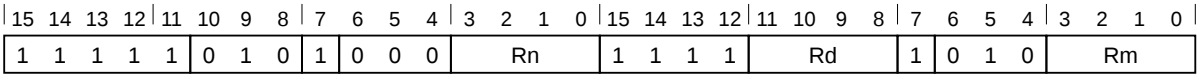
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
4      diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
5      bits(32) result;
6      result<15:0>  = SignedSat(sum,  16);
7      result<31:16> = SignedSat(diff, 16);
8      R[d] = result;
```

## C2.4.143 QSUB

Saturating Subtract. Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range $-2^{31}$ to $2^{31}-1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rn> | Is the second general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

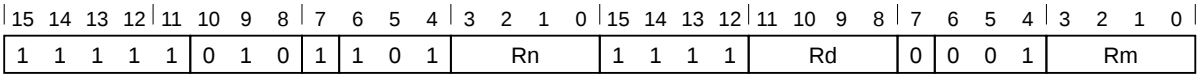
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
4      if sat then
5          APSR.Q = '1';
```

## C2.4.144 QSUB16

Saturating Subtract 16. Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range $-2^{15}$ to $2^{15}-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

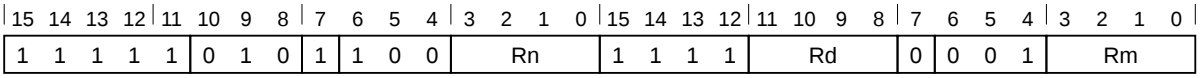
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
4      diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
5      bits(32) result;
6      result<15:0>  = SignedSat(diff1, 16);
7      result<31:16> = SignedSat(diff2, 16);
8      R[d] = result;
```

## C2.4.145 QSUB8

Saturating Subtract 8. Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range $-2^7$ to $2^7-1$, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

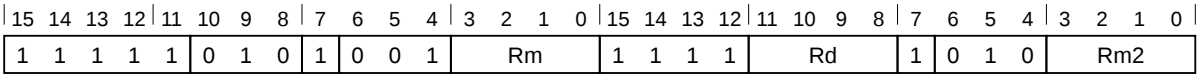
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
4      diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
5      diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
6      diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
7      R[d]<7:0>   = SignedSat(diff1, 8);
8      R[d]<15:8>  = SignedSat(diff2, 8);
9      R[d]<23:16> = SignedSat(diff3, 8);
10     R[d]<31:24> = SignedSat(diff4, 8);
```

## C2.4.146 RBIT

Reverse Bits. Reverse Bits reverses the bit order in a 32-bit register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`RBIT{<c>}{<q>} <Rd>, <Rm>`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if Rm != Rm2 then UNPREDICTABLE;
3  d = UInt(Rd);  m = UInt(Rm);
4  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `Rm != Rm2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

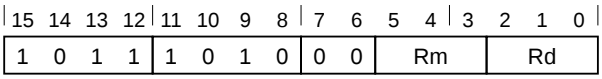
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      bits(32) result;
4      for i = 0 to 31
5          result<31-i> = R[m]<i>;
6      R[d] = result;
```

## C2.4.147 REV

Byte-Reverse Word. Byte-Reverse Word reverses the byte order in a 32-bit register.

### T1

*Armv8-M*




#### T1 variant

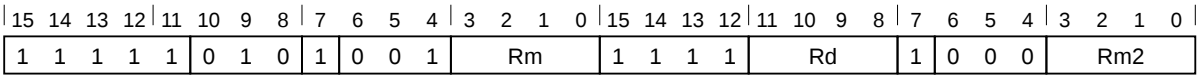
```
REV{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1 d = UInt(Rd);  m = UInt(Rm);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
REV{<c>}.W <Rd>, <Rm>
  // <Rd>, <Rm> can be represented in T1
REV{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 if Rm != Rm2 then UNPREDICTABLE;
3 d = UInt(Rd);  m = UInt(Rm);
4 if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `Rm != Rm2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | For encoding T1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

### Operation for all encodings

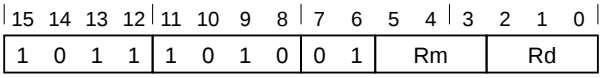
```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

## C2.4.148 REV16

Byte-Reverse Packed Halfword. Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

### T1

*Armv8-M*




#### T1 variant

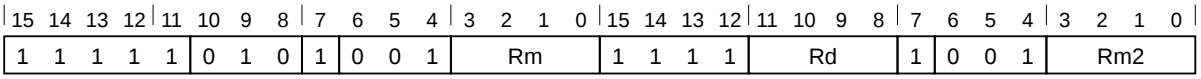
```
REV16{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  m = UInt(Rm);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
REV16{<c>}.W <Rd>, <Rm>
  // <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if Rm != Rm2 then UNPREDICTABLE;
3  d = UInt(Rd);  m = UInt(Rm);
4  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `Rm != Rm2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<Rm>` | For encoding T1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

## Operation for all encodings

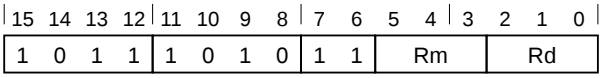
```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

## C2.4.149 REVSH

Byte-Reverse Signed Halfword. Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32 bits.

### T1

*Armv8-M*




#### T1 variant

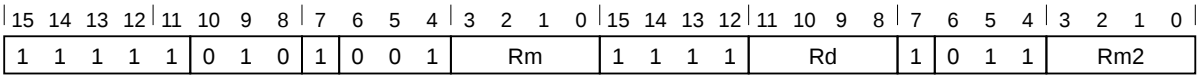
```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  m = UInt(Rm);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
REVSH{<c>}.W <Rd>, <Rm>
  // <Rd>, <Rm> can be represented in T1
REVSH{<c>}{<q>} <Rd>, <Rm>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if Rm != Rm2 then UNPREDICTABLE;
3  d = UInt(Rd);  m = UInt(Rm);
4  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `Rm != Rm2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | For encoding T1: is the general-purpose source register, encoded in the "Rm" field.<br>For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice. |

## Operation for all encodings

```
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;
```

## C2.4.150 ROR (immediate)

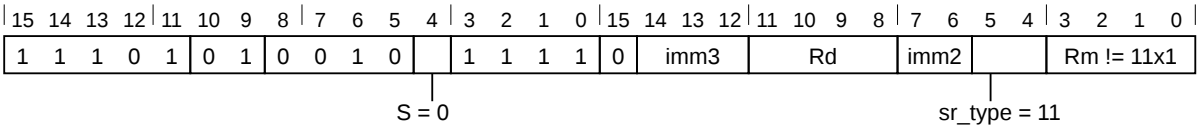
Rotate Right (immediate). Rotate Right (immediate) rotates a register value by a constant number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T3

*Armv8-M Main Extension only*




**MOV, shift or rotate by value variant**

**Applies when `!(imm3 == 000 && imm2 == 00)`.**

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | Is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

### Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.151 ROR (register)

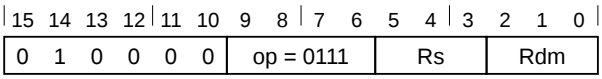
Rotate Right (register). Rotate Right (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M Main Extension only*




**Rotate right variant**

```
ROR<c>{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Inside IT block
```

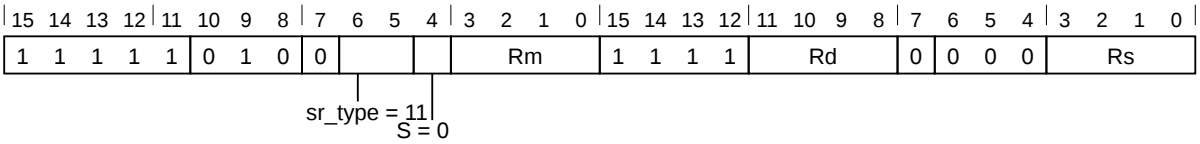
**is equivalent to**

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

**and is the preferred disassembly when `InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Non flag setting variant**

```
ROR<c>.W {<Rd>,} <Rm>, <Rs>
  // Inside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

**Non flag setting variant**

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.152 RORS (immediate)

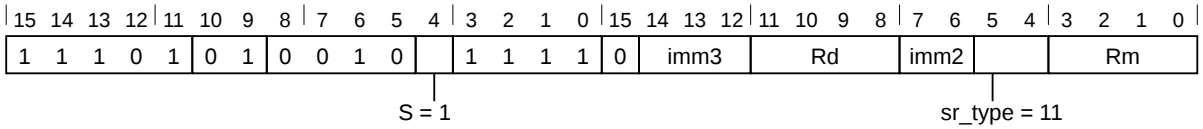
Rotate Right, Setting flags (immediate). Rotate Right, Setting flags (immediate) rotates a register value by a constant number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T3

*Armv8-M Main Extension only*




**MOVS, shift or rotate by value variant**

**Applies when `!(imm3 == 000 && imm2 == 00)`.**

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR #<imm>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <imm> | Is the shift amount, in the range 1 to 31, encoded in the "imm3:imm2" field. |

### Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.153 RORS (register)

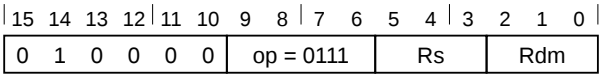
Rotate Right, Setting flags (register). Rotate Right, Setting flags (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

### T1

*Armv8-M*




**Rotate right variant**

```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs>
  // Outside IT block
```

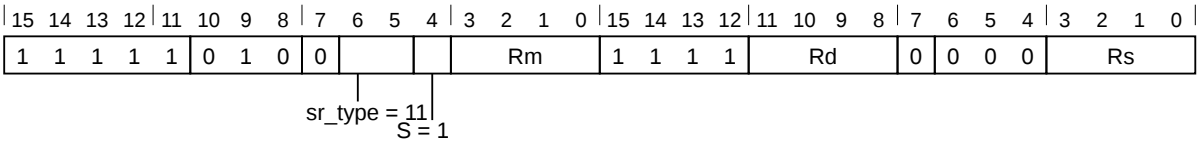
**is equivalent to**

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

**and is the preferred disassembly when `!InITBlock()`.**

### T2

*Armv8-M Main Extension only*




**Flag setting variant**

```
RORS.W {<Rd>,} <Rm>, <Rs>
  // Outside IT block, and <Rd>, <Rm>, <sr_type>, <Rs> can be represented in T1
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

**Flag setting variant**

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdm> | Is the first general-purpose source register and the destination register, encoded in the "Rdm" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the first general-purpose source register, encoded in the "Rm" field. |
| <Rs> | Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field. |

### Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

## C2.4.154 RRX

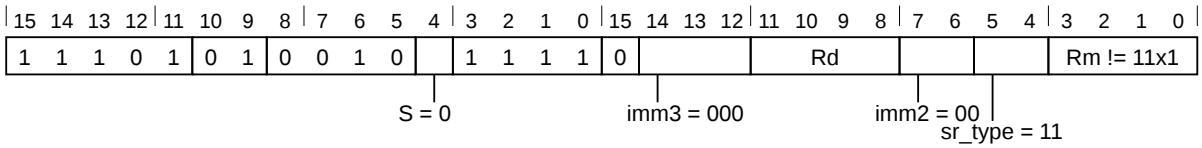
Rotate Right with Extend. Rotate Right with Extend shifts a register value right by one bit, shifting the Carry flag into bit[31], and writes the result to the destination register.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T3

*Armv8-M Main Extension only*




**MOV, rotate right with extend variant**

```
RRX{<c>}{<q>} {<Rd>,} <Rm>
```

**is equivalent to**

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.155 RRXS

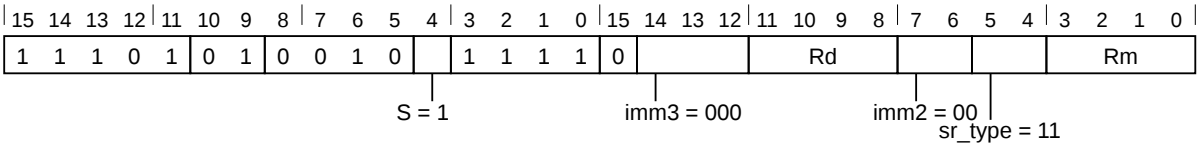
Rotate Right with Extend, Setting flags. Rotate Right with Extend, Setting flags shifts a register value right by one bit, shifting the Carry flag into bit[31] and bit[0] into the Carry flag, writes the result to the destination register and updates the condition flags (other than Carry) based on the result.

This instruction is an alias of the MOV (register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV (register).
- The description of MOV (register) gives the operational pseudocode for this instruction.

### T3

*Armv8-M Main Extension only*




**MOVS, rotate right with extend variant**

```
RRXS{<c>}{<q>} {<Rd>,} <Rm>
```

**is equivalent to**

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

## C2.4.156 RSB (immediate)

Reverse Subtract (immediate). Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*


#### T1 variant

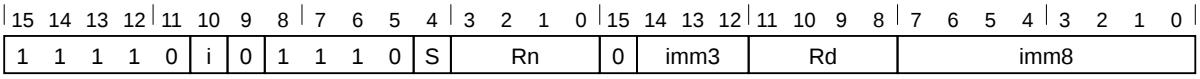
```
RSB<c>{<q>} {<Rd>, }<Rn>, #0
  // Inside IT block
RSBS{<q>} {<Rd>, }<Rn>, #0
  // Outside IT block
```

### Decode for this encoding

```
1  d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

### T2

*Armv8-M Main Extension only*




#### RSB variant

**Applies when `S == 0`.**

```
RSB<c>.W {<Rd>,} <Rn>, #0
  // Inside IT block
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

#### RSBS variant

**Applies when `S == 1`.**

```
RSBS.W {<Rd>,} <Rn>, #0
  // Outside IT block
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
3  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |

| | |
|---|---|
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

## Operation for all encodings

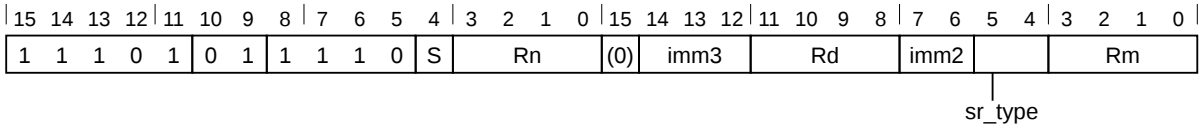
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

## C2.4.157 RSB (register)

Reverse Subtract (register). Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M Main Extension only*




**RSB, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**RSB, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**RSBS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**RSBS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
3  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

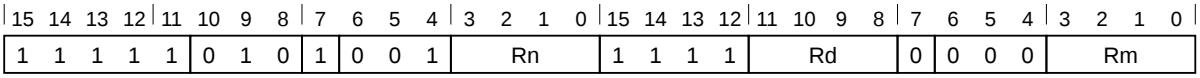
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        APSR.Z = IsZeroBit(result);
        APSR.N = result<31>;
        APSR.C = carry;
        APSR.V = overflow;
```

## C2.4.158 SADD16

Signed Add 16. Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

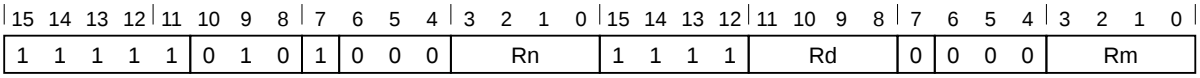
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
4      sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
5      R[d] = sum2<15:0> : sum1<15:0>;
6      APSR.GE<1:0> = if sum1 >= 0 then '11' else '00';
7      APSR.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

## C2.4.159 SADD8

Signed Add 8. Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

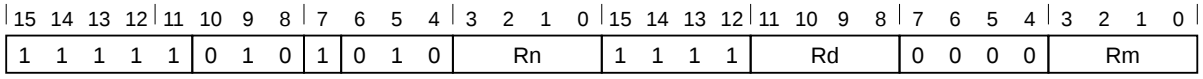
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
4      sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
5      sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
6      sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
7      R[d] = sum4<7:0> : sum3<7:0> : sum2<7:0> : sum1<7:0>;
8      APSR.GE<0>  = if sum1 >= 0 then '1' else '0';
9      APSR.GE<1>  = if sum2 >= 0 then '1' else '0';
10     APSR.GE<2>  = if sum3 >= 0 then '1' else '0';
11     APSR.GE<3>  = if sum4 >= 0 then '1' else '0';
```

## C2.4.160 SASX

Signed Add and Subtract with Exchange. Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

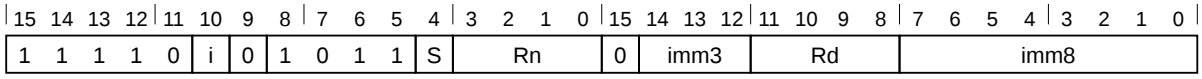
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
4      sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
5      R[d] = sum<15:0> : diff<15:0>;
6      APSR.GE<1:0> = if diff >= 0 then '11' else '00';
7      APSR.GE<3:2> = if sum  >= 0 then '11' else '00';
```

## C2.4.161 SBC (immediate)

Subtract with Carry (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.

### T1

*Armv8-M Main Extension only*




**SBC variant**

**Applies when `S == 0`.**

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**SBCS variant**

**Applies when `S == 1`.**

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
3  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

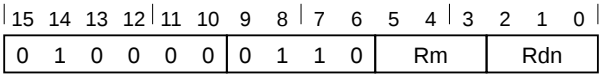
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
4      R[d] = result;
5      if setflags then
6          APSR.N = result<31>;
7          APSR.Z = IsZeroBit(result);
8          APSR.C = carry;
9          APSR.V = overflow;
```

## C2.4.162 SBC (register)

Subtract with Carry (register). Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

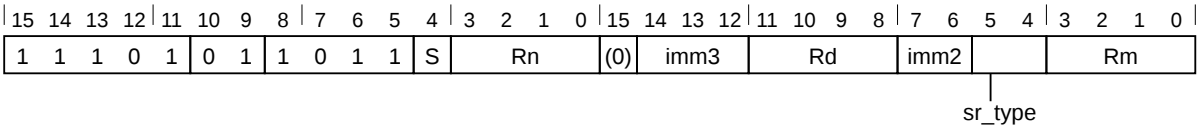
```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Inside IT block
SBCS{<q>} {<Rdn>,} <Rdn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




**SBC, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**SBC, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
SBC<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**SBCS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**SBCS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
SBCS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the first general-purpose source register and the destination register, encoded in the "Rdn" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

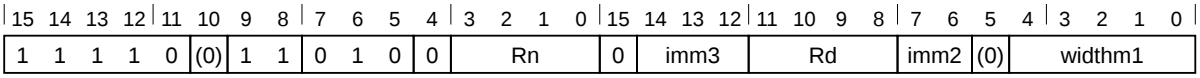
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

## C2.4.163 SBFX

Signed Bit Field Extract. Signed Bit Field Extract extracts any number of adjacent bits at any position from one register, sign extends them to 32 bits, and writes the result to the destination register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);
3  lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
4  msbit = lsbit + widthminus1;
5  if msbit > 31 then UNPREDICTABLE;
6  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | Is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if msbit <= 31 then
4          R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
5      else
6          R[d] = bits(32) UNKNOWN;
```

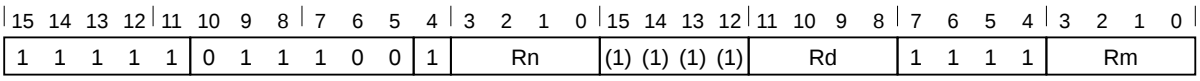
## C2.4.164 SDIV

Signed Divide. Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value and writes the result to the destination register. The condition code flags are not affected.

If R[n] == 0x80000000 ($-2^{31}$) and R[m] == 0xFFFFFFFF (-1), the result of the division is 0x80000000.

### T1

*Armv8-M*




#### T1 variant

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
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 symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the dividend, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the divisor, encoded in the "Rm" field. |

### Operation for all encodings

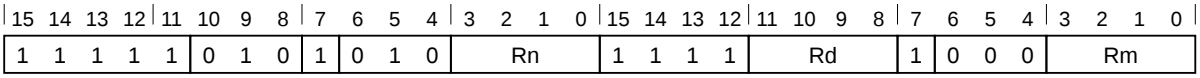
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if SInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```

## C2.4.165 SEL

Select Bytes. Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      bits(32) result;
4      result<7:0>   = if APSR.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
5      result<15:8>  = if APSR.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
6      result<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
7      result<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
8      R[d] = result;
```

## C2.4.166 SEV

Send Event. Send Event is a hint instruction. It causes an event to be signaled to all PEs within the multiprocessor system.

This is a NOP-compatible hint.

### T1

*Armv8-M*

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

#### T1 variant

```
SEV{<c>}{<q>}
```

### Decode for this encoding

```
// No additional decoding required
```

### T2

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

#### T2 variant

```
SEV{<c>}.W
```

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

## C2.4.167 SG

Secure Gateway. Secure Gateway marks a valid branch target for branches from Non-secure code that call Secure code.

This instruction sets the Security state to Secure if its address is in Secure memory. If the address of this instruction is in Non-secure memory, the instruction behaves as a NOP.

If the PE was previously in Non-secure state:

- This instruction sets bit[0] of LR to 0, to indicate that the return address will cause a transition from Secure to Non-secure state.
- If the Floating-point Extension is implemented, this instruction marks Secure floating-point state as inactive, by setting CONTROL_S.SFPA to 0. This indicates that the floating-point registers do not contain active state that belongs to the Secure state.

An INVEP SecureFault is generated if the PE attempts to reenter Thread mode when CCR_S.TRD is set to 1 and either or both of the following are true:

- CONTROL_S.SPSEL is 0.
- The current Stack Pointer of the Secure state points to an address that contains the value `0xFEFA125A`[31:1].

If the Security Extension is not implemented, this instruction behaves as a NOP.

SG is an unconditional instruction and executes as such both inside and outside an IT instruction block. Arm recommends that software does not place SG inside an IT instruction block.

### T1

*Armv8-M*

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

#### T1 variant

```
SG{<q>}
```

### Decode for this encoding

```
// No encoding specific operations
```

### Assembler symbols for all encodings

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
EncodingSpecificOperations();

if HaveSecurityExt() then
    sAttributes = SecurityCheck(ThisInstrAddr(), TRUE, IsSecure());
    if !sAttributes.ns then
        if !IsSecure() then
            if HasArchVersion(Armv8p1)  && CurrentMode() == PEMode_Thread then
                // The access to the Secure stack should be performed with the priviledge
                // level of the current mode in the Secure state, and not the current state.
                // NOTE: The load below is performed, and any faults handled even if thread
                //       mode re-entrancy checking is disabled.
                secIsPriv     = CurrentModeIsPrivileged(TRUE);
                secSp         = LookUpSP_with_security_mode(TRUE, PEMode_Thread);
```

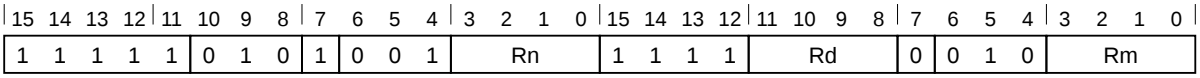
```
14                (exc, spData) = MemA_with_priv_security(_SP(secSp), 4,    AccType_NORMAL,
15                                                        secIsPriv,  TRUE, TRUE);
16                HandleException(exc);
17                // Check for an exception stack frame if thread mode re-entrancy is disabled.
18                if CCR_S.TRD == '1' && (spData<31:1> == 0xFEFA125A<31:1> ||
19                                        secSp        == RNamesSP_Main_Secure) then
20                    SFSR.INVEP = '1';
21                    HandleException(CreateException(SecureFault));
22
23            // Set up the security meta data flags and change to the Secure state
24            LR<0> = '0';
25            if HaveMveOrFPExt() then
26                CONTROL_S.SFPA = '0';
27            // LOB data cleared to prevent Non-secure code from interfering
28            // with Secure execution
29            if HaveLOBExt() then
30                LO_BRANCH_INFO.VALID = '0';
31        CurrentState = SecurityState_Secure;
32        // IT/ICI/ECI data cleared to prevent Non-secure code from interfering
33        // with Secure execution
34        if HaveMainExt() then
35            ITSTATE = Zeros(8);
```

## C2.4.168 SHADD16

Signed Halving Add 16. Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

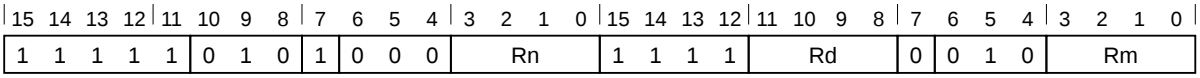
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
4      sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
5      R[d] = sum2<16:1> : sum1<16:1>;
```

## C2.4.169 SHADD8

Signed Halving Add 8. Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

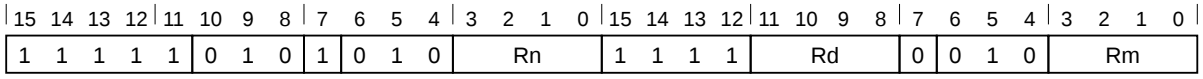
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
4      sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
5      sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
6      sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
7      R[d] = sum4<8:1> : sum3<8:1> : sum2<8:1> : sum1<8:1>;
```

## C2.4.170 SHASX

Signed Halving Add and Subtract with Exchange. Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

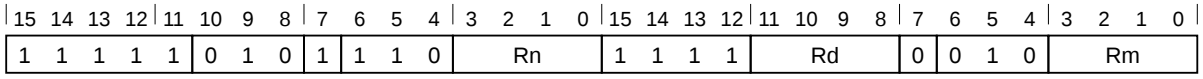
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
4      sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
5      R[d] = sum<16:1> : diff<16:1>;
```

## C2.4.171 SHSAX

Signed Halving Subtract and Add with Exchange. Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

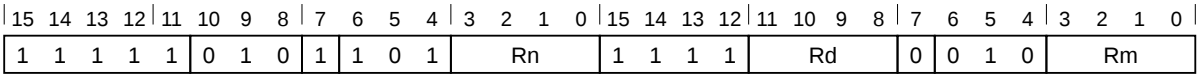
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
4      diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
5      R[d] = diff<16:1> : sum<16:1>;
```

## C2.4.172 SHSUB16

Signed Halving Subtract 16. Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

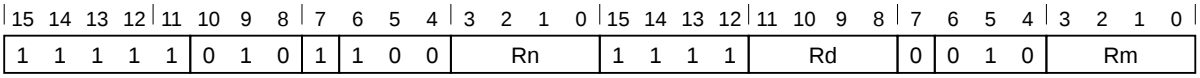
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
4      diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
5      R[d]  = diff2<16:1> : diff1<16:1>;
```

## C2.4.173 SHSUB8

Signed Halving Subtract 8. Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
4      diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
5      diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
6      diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
7      R[d]  = diff4<8:1> : diff3<8:1> : diff2<8:1> : diff1<8:1>;
```

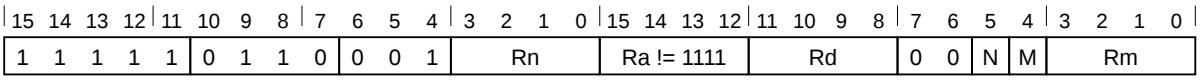
## C2.4.174 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords). Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. It is not possible for overflow to occur during the multiplication.

### T1

*Armv8-M DSP Extension only*




**SMLABB variant**

**Applies when `N == 0 && M == 0`.**

```
SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLABT variant**

**Applies when `N == 0 && M == 1`.**

```
SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLATB variant**

**Applies when `N == 1 && M == 0`.**

```
SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLATT variant**

**Applies when `N == 1 && M == 1`.**

```
SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
4  n_high = (N == '1');  m_high = (M == '1');
5  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

## C2.4.175 SMLAD, SMLADX

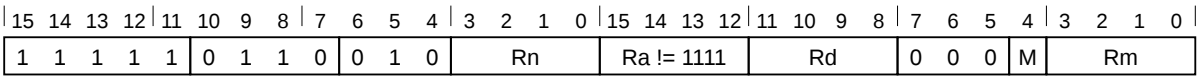
Signed Multiply Accumulate Dual. Signed Multiply Accumulate Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

### T1

*Armv8-M DSP Extension only*




**SMLAD variant**

**Applies when `M == 0`.**

```
SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLADX variant**

**Applies when `M == 1`.**

```
SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE SMUAD;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
4  m_swap = (M == '1');
5  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

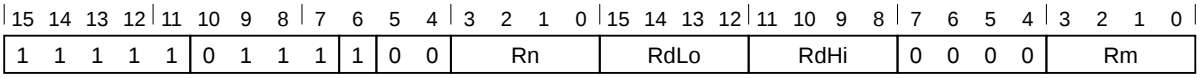
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand2 = if m_swap then ROR(R[m],16) else R[m];
4      product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
5      product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
6      result = product1 + product2 + SInt(R[a]);
7      R[d] = result<31:0>;
8      if result != SInt(result<31:0>) then  // Signed overflow
9          APSR.Q = '1';
```

## C2.4.176 SMLAL

Signed Multiply Accumulate Long. Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<RdLo>` | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| `<RdHi>` | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| `<Rn>` | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| `<Rm>` | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
4      R[dHi] = result<63:32>;
5      R[dLo] = result<31:0>;
```

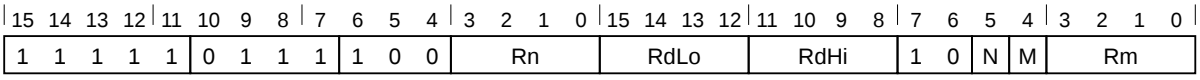
## C2.4.177 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords). Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### T1

*Armv8-M DSP Extension only*




**SMLALBB variant**

**Applies when `N == 0 && M == 0`.**

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

**SMLALBT variant**

**Applies when `N == 0 && M == 1`.**

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

**SMLALTB variant**

**Applies when `N == 1 && M == 0`.**

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

**SMLALTT variant**

**Applies when `N == 1 && M == 1`.**

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
3  n_high = (N == '1');  m_high = (M == '1');
4  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
5  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <x>), encoded in the "Rm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

## C2.4.178 SMLALD, SMLALDX

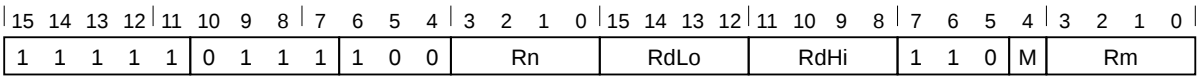
Signed Multiply Accumulate Long Dual. Signed Multiply Accumulate Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### T1

*Armv8-M DSP Extension only*




**SMLALD variant**

**Applies when `M == 0`.**

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

**SMLALDX variant**

**Applies when `M == 1`.**

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
```

```
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

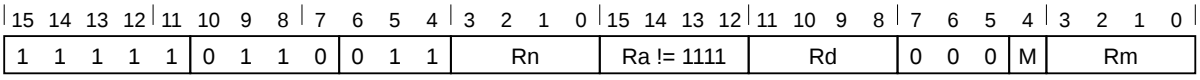
## C2.4.179 SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword). Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. No overflow can occur during the multiplication.

### T1

*Armv8-M DSP Extension only*




**SMLAWB variant**

**Applies when `M == 0`.**

```
SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLAWT variant**

**Applies when `M == 1`.**

```
SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE "SMULWB, SMULWT";
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
4      result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
5      R[d] = result<47:16>;
6      if (result >> 16) != SInt(R[d]) then  // Signed overflow
7          APSR.Q = '1';
```

## C2.4.180 SMLSD, SMLSDX

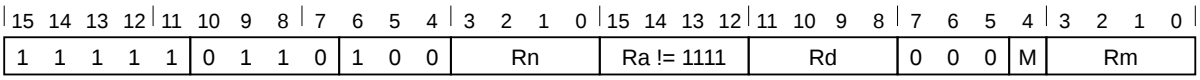
Signed Multiply Subtract Dual. Signed Multiply Subtract Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

### T1

*Armv8-M DSP Extension only*




**SMLSD variant**

**Applies when `M == 0`.**

```
SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMLSDX variant**

**Applies when `M == 1`.**

```
SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE SMUSD;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand2 = if m_swap then ROR(R[m],16) else R[m];
4      product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
5      product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
6      result = product1 - product2 + SInt(R[a]);
7      R[d] = result<31:0>;
8      if result != SInt(result<31:0>) then  // Signed overflow
9          APSR.Q = '1';
```

## C2.4.181 SMLSLD, SMLSLDX

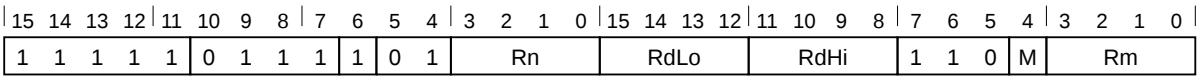
Signed Multiply Subtract Long Dual. Signed Multiply Subtract Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo $2^{64}$.

### T1

*Armv8-M DSP Extension only*




**SMLSLD variant**

**Applies when `M == 0`.**

```
SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

**SMLSLDX variant**

**Applies when `M == 1`.**

```
SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
```

```
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

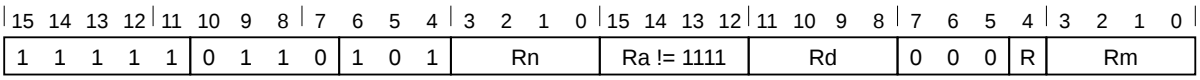
## C2.4.182 SMMLA, SMMLAR

Signed Most Significant Word Multiply Accumulate. Signed Most Significant Word Multiply Accumulate multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

### T1

*Armv8-M DSP Extension only*




**SMMLA variant**

**Applies when `R == 0`.**

```
SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMMLAR variant**

**Applies when `R == 1`.**

```
SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE SMMUL;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
4      if round then result = result + 0x80000000;
5      R[d] = result<63:32>;
```

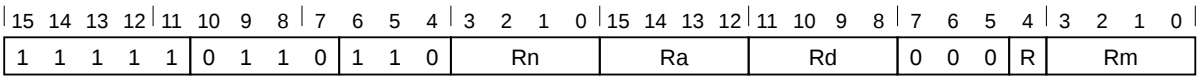
## C2.4.183 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract. Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

### T1

*Armv8-M DSP Extension only*




**SMMLS variant**

**Applies when `R == 0`.**

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

**SMMLSR variant**

**Applies when `R == 1`.**

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
4     if round then result = result + 0x80000000;
5     R[d] = result<63:32>;
```

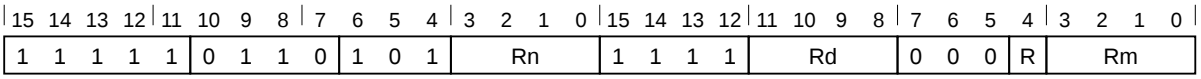
## C2.4.184 SMMUL, SMMULR

Signed Most Significant Word Multiply. Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

### T1

*Armv8-M DSP Extension only*




**SMMUL variant**

**Applies when `R == 0`.**

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMMULR variant**

**Applies when `R == 1`.**

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = SInt(R[n]) * SInt(R[m]);
4      if round then result = result + 0x80000000;
5      R[d] = result<63:32>;
```

## C2.4.185 SMUAD, SMUADX

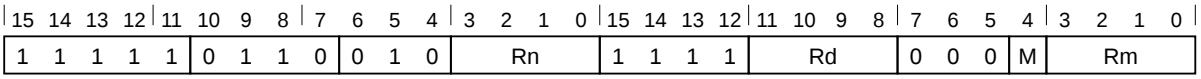
Signed Dual Multiply Add. Signed Dual Multiply Add performs two signed 16-bit by 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

### T1

*Armv8-M DSP Extension only*




**SMUAD variant**

**Applies when `M == 0`.**

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMUADX variant**

**Applies when `M == 1`.**

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

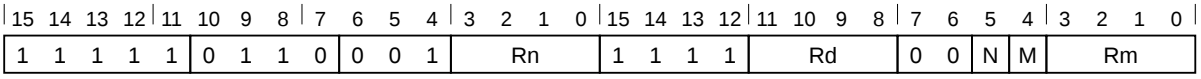
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand2 = if m_swap then ROR(R[m],16) else R[m];
4      product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
5      product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
6      result = product1 + product2;
7      R[d] = result<31:0>;
8      if result != SInt(result<31:0>) then  // Signed overflow
9          APSR.Q = '1';
```

## C2.4.186 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords). Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

### T1

*Armv8-M DSP Extension only*




**SMULBB variant**

**Applies when `N == 0 && M == 0`.**

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMULBT variant**

**Applies when `N == 0 && M == 1`.**

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMULTB variant**

**Applies when `N == 1 && M == 0`.**

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMULTT variant**

**Applies when `N == 1 && M == 1`.**

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  n_high = (N == '1');  m_high = (M == '1');
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand in the bottom or top half (selected by <x>), encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
4      operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
```

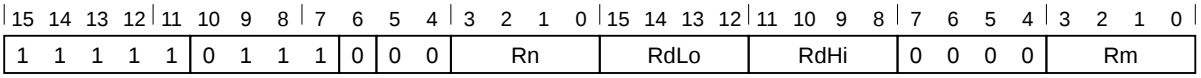
```
5       result = SInt(operand1) * SInt(operand2);
6       R[d] = result<31:0>;
7       // Signed overflow cannot occur
```

## C2.4.187 SMULL

Signed Multiply Long. Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation for all encodings

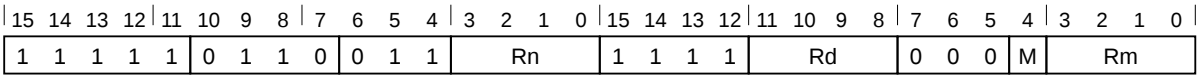
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = SInt(R[n]) * SInt(R[m]);
4      R[dHi] = result<63:32>;
5      R[dLo] = result<31:0>;
```

## C2.4.188 SMULWB, SMULWT

Signed Multiply (word by halfword). Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

### T1

*Armv8-M DSP Extension only*




**SMULWB variant**

**Applies when `M == 0`.**

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMULWT variant**

**Applies when `M == 1`.**

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier in the bottom or top half (selected by <y>), encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
4      product = SInt(R[n]) * SInt(operand2);
5      R[d] = product<47:16>;
6      // Signed overflow cannot occur
```

## C2.4.189 SMUSD, SMUSDX

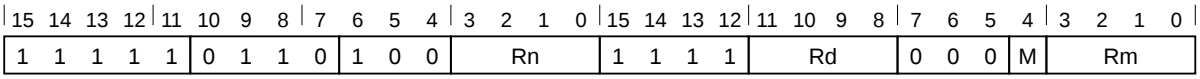
Signed Dual Multiply Subtract. Signed Dual Multiply Subtract performs two signed 16-bit by 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow cannot occur.

### T1

*Armv8-M DSP Extension only*




**SMUSD variant**

**Applies when `M == 0`.**

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

**SMUSDX variant**

**Applies when `M == 1`.**

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1 if !HaveDSPExt() then UNDEFINED;
2 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
3 if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

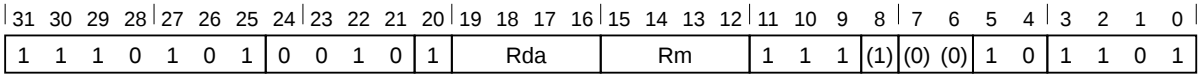
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     operand2 = if m_swap then ROR(R[m],16) else R[m];
4     product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
5     product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
6     result = product1 - product2;
7     R[d] = result<31:0>;
8     // Signed overflow cannot occur
```

## C2.4.190 SQRSHR (register)

Signed Saturating Rounding Shift Right. Signed saturating rounding shift right by 0 to 32 bits of a 32 bit value stored in a general-purpose register. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: SQRSHR variant

```
SQRSHR<c> Rda, Rm
```

### Decode for this encoding

```
1  if !HaveMainExt()            then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3  if !HaveMve()                then UNDEFINED;
4  da = UInt(Rda);
5  m  = UInt(Rm);
6  if Rda == '11x1' || Rm == '11x1' || Rm == Rda then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

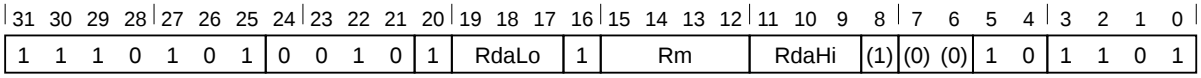
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      amount = SInt(R[m]<7:0>);
5      op1    = SInt(R[da]);
6      op1    = op1 + (1 << (amount - 1));
7      (result, sat) = SignedSatQ((op1 >> amount), 32);
8      if sat then APSR.Q = '1';
9      R[da] = result<31:0>;
```

## C2.4.191 SQRSHRL (register)

Signed Saturating Rounding Shift Right Long. Signed saturating rounding shift right by 0 to 64 bits of a 64 bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: SQRSHRL variant

```
SQRSHRL<c> RdaLo, RdaHi, Rm
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "SQRSHR (register)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  m   = UInt(Rm);
8  if RdaHi == '110' || Rm == '11x1' || Rm == RdaHi:'1' then CONSTRAINED_UNPREDICTABLE;
9  if Rm    == RdaLo:'0' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

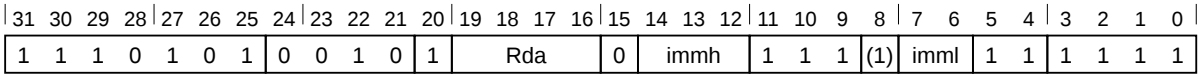
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      amount = SInt(R[m]<7:0>);
5      op1    = SInt(R[dah]:R[dal]);
6      op1    = op1 + (1 << (amount - 1));
7      (result, sat) = SignedSatQ((op1 >> amount), 64);
8      if sat then APSR.Q = '1';
9      R[dah] = result<63:32>;
10     R[dal] = result<31:0>;
```

## C2.4.192 SQSHL (immediate)

Signed Saturating Shift Left. Signed saturating shift left by 1 to 32 bits of a 32 bit value stored in a general-purpose register.

### T1

*Armv8.1-M MVE*




#### T1: SQSHL variant

```
SQSHL<c> Rda, #<imm>
```

### Decode for this encoding

```
1  if !HaveMainExt()           then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3  if !HaveMve()               then UNDEFINED;
4  da = UInt(Rda);
5  (-, amount) = DecodeImmShift('10', immh:imml);
6  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

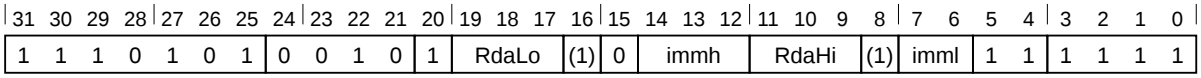
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1 = SInt(R[da]);
5      (result, sat) = SignedSatQ((op1 << amount), 32);
6      if sat then APSR.Q = '1';
7      R[da] = result<31:0>;
```

## C2.4.193 SQSHLL (immediate)

Signed Saturating Shift Left Long. Signed saturating shift left by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: SQSHLL variant

```
SQSHLL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "SQSHL (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

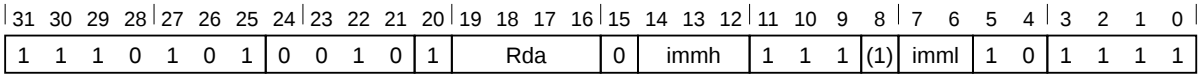
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1 = SInt(R[dah]:R[dal]);
5      (result, sat) = SignedSatQ((op1 << amount), 64);
6      if sat then APSR.Q = '1';
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

## C2.4.194 SRSHR (immediate)

Signed Rounding Shift Right. Signed rounding shift right by 1 to 32 bits of a 32 bit value stored in a general-purpose register.

### T1

*Armv8.1-M MVE*




#### T1: SRSHR variant

```
SRSHR<c> Rda, #<imm>
```

### Decode for this encoding

```
1  if !HaveMainExt()           then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3  if !HaveMve()               then UNDEFINED;
4  da = UInt(Rda);
5  (-, amount) = DecodeImmShift('10', immh:imml);
6  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

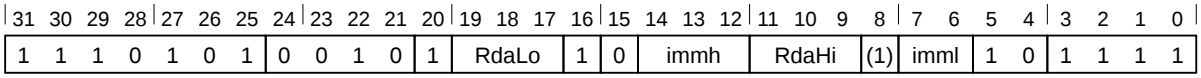
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = SInt(R[da]);
5      op1    = op1 + (1 << (amount - 1));
6      result = (op1 >> amount)<31:0>;
7      R[da]  = result<31:0>;
```

## C2.4.195 SRSHRL (immediate)

Signed Rounding Shift Right Long. Signed rounding shift right by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: SRSHRL variant

```
SRSHRL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "SRSHR (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = SInt(R[dah]:R[dal]);
5      op1    = op1 + (1 << (amount - 1));
6      result = (op1 >> amount)<63:0>;
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

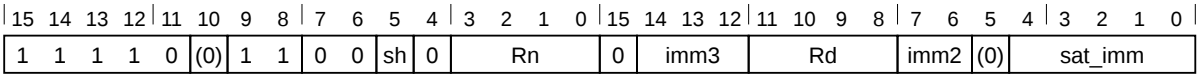
## C2.4.196 SSAT

Signed Saturate. Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

The APSR.Q flag is set to 1 if the operation saturates.

### T1

*Armv8-M Main Extension only*




**Arithmetic shift right variant**

**Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.**

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

**Logical shift left variant**

**Applies when `sh == 0`.**

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

### Decode for this encoding

```
1 if sh == '1' && (imm3:imm2) == '00000' then
2     if HaveDSPExt() then
3         SEE SSAT16;
4     else
5         UNDEFINED;
6 if !HaveMainExt() then UNDEFINED;
7 d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
8 (shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
9 if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 1 to 32, encoded in the "sat_imm" field as <imm>-1. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <amount> | For the arithmetic shift right variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.<br>For the logical shift left variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
4     (result, sat) = SignedSatQ(SInt(operand), saturate_to);
5     R[d] = SignExtend(result, 32);
6     if sat then
7         APSR.Q = '1';
```

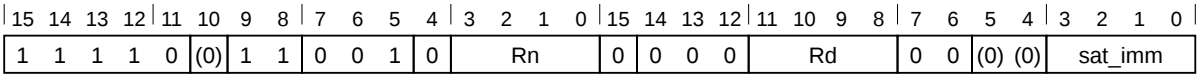
## C2.4.197 SSAT16

Signed Saturate 16. Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

The APSR.Q flag is set to 1 if the operation saturates.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
3  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 1 to 16, encoded in the "sat_imm" field as <imm>-1. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

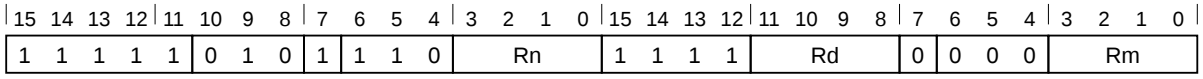
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
4      (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
5      bits(32) result;
6      result<15:0>  = SignExtend(result1, 16);
7      result<31:16> = SignExtend(result2, 16);
8      R[d] = result;
9      if sat1 || sat2 then
10         APSR.Q = '1';
```

## C2.4.198 SSAX

Signed Subtract and Add with Exchange. Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
4      diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
5      R[d] = diff<15:0> : sum<15:0>;
6      APSR.GE<1:0> = if sum  >= 0 then '11' else '00';
7      APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

## C2.4.199 SSBB

Speculative Store Bypass Barrier. Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
  - The store is to the same location as the load.
  - The store appears in program order after the SSBB.

### T1

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (1) | (1) | (1) | (1) | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

#### T1 variant

`SSBB{<q>}`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  if InITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

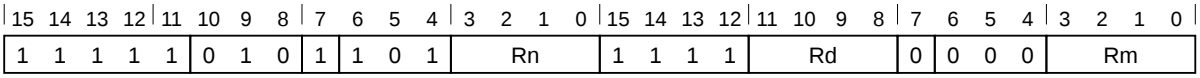
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      SpeculativeSynchronizationBarrier();
```

## C2.4.200 SSUB16

Signed Subtract 16. Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

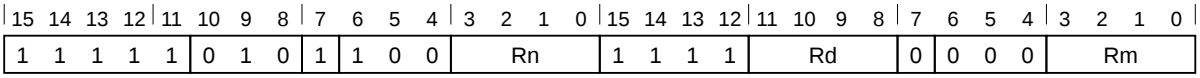
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
4      diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
5      R[d]  = diff2<15:0> : diff1<15:0>;
6      APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
7      APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

## C2.4.201 SSUB8

Signed Subtract 8. Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
4      diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
5      diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
6      diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
7      R[d]  = diff4<7:0> : diff3<7:0> : diff2<7:0> : diff1<7:0>;
8      APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
9      APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
10     APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
11     APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

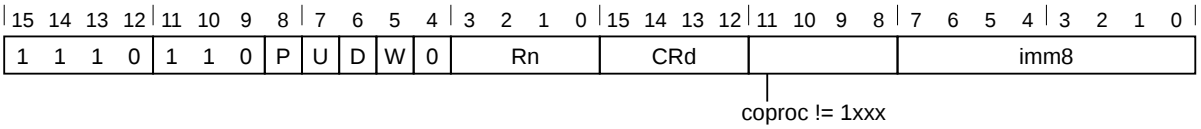
## C2.4.202 STC, STC2

Store Coprocessor. Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

### T1

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `P == 1 && W == 0`.**

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

#### Pre-indexed variant

**Applies when `P == 1 && W == 1`.**

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

#### Unindexed variant

**Applies when `P == 0 && U == 1 && W == 0`.**

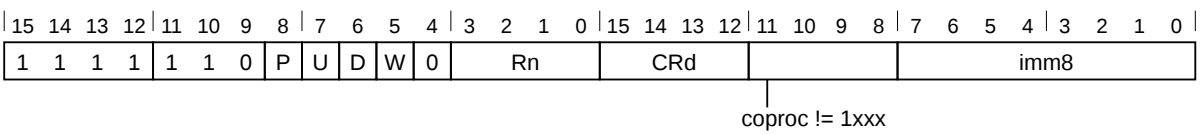
```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

### Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}           then SEE "Floating-point and MVE";
2  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
3  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
4  if !HaveMainExt() then UNDEFINED;
5  n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
6  index = (P == '1');  add = (U == '1');  wback = (W == '1');
7  if n == 15 then UNPREDICTABLE;
```

### T2

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `P == 1 && W == 0`.**

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

**Unindexed variant**

**Applies when `P == 0 && U == 1 && W == 0`.**

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

## Decode for this encoding

```
if coproc IN {'100x', '101x', '111x'}           then SEE "Floating-point and MVE";
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
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 symbols for all encodings

| | |
|---|---|
| L | If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form. |
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <coproc> | Is the name of the coprocessor, encoded in the "coproc" field. The valid coprocessor names are p10, p11, p14, and p15. |
| <CRd> | Is the coprocessor register to be transferred, encoded in the "CRd" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| <option> | Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteCPCheck(cp);
    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];

        // Determine if the stack pointer limit check should be performed
        if wback && n == 13 then
            violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
```

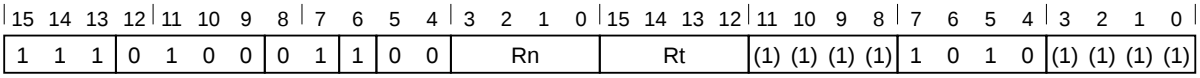
```
13          else
14              violatesLimit = FALSE;
15
16          // Memory operation only performed if limit not violated
17          if !violatesLimit then
18              repeat
19                  MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());
20                  address         = address + 4;
21              until Coproc_DoneStoring(cp, ThisInstr());
22
23          // If the stack pointer is being updated a fault will be raised
24          // if the limit is violated
25          if wback then RSPCheck[n] = offset_addr;
```

## C2.4.203 STL

Store-Release Word. Store Release Word stores a word from a register to memory. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt); n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

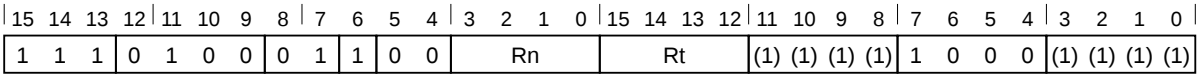
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      MemO[address, 4] = R[t];
```

## C2.4.204 STLB

Store-Release Byte. Store Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt); n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

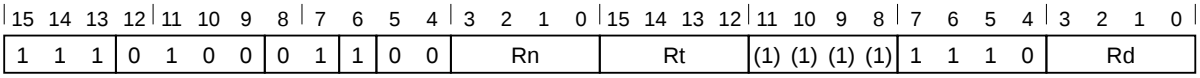
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      MemO[address, 1] = R[t]<7:0>;
```

## C2.4.205 STLEX

Store-Release Exclusive Word. Store Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
2  if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>`1` If the operation fails to update memory.<br>`0` If the operation updates memory. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      if ExclusiveMonitorsPass(address,4) then
5          MemO[address, 4] = R[t];
6          R[d] = ZeroExtend('0');
```

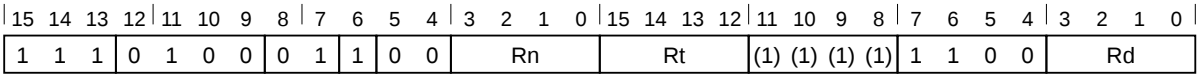
```
7       else
8           R[d] = ZeroExtend('1');
```

## C2.4.206 STLEXB

Store-Release Exclusive Byte. Store Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
2  if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>1 If the operation fails to update memory.<br>0 If the operation updates memory. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      if ExclusiveMonitorsPass(address,1) then
5          MemO[address, 1] = R[t]<7:0>;
6          R[d] = ZeroExtend('0');
```

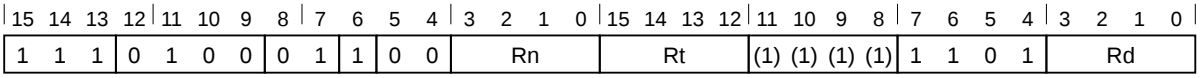
```
7       else
8           R[d] = ZeroExtend('1');
```

## C2.4.207 STLEXH

Store-Release Exclusive Halfword. Store Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
2  if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>`1` If the operation fails to update memory.<br>`0` If the operation updates memory. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      if ExclusiveMonitorsPass(address,2) then
5          MemO[address, 2] = R[t]<15:0>;
```

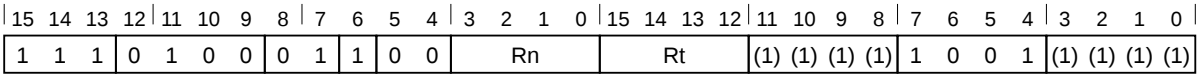
```
6           R[d] = ZeroExtend('0');
7       else
8           R[d] = ZeroExtend('1');
```

## C2.4.208 STLH

Store-Release Halfword. Store Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics.

### T1

*Armv8-M*




#### T1 variant

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  t = UInt(Rt); n = UInt(Rn);
2  if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

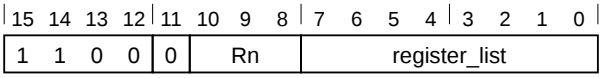
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      MemO[address, 2] = R[t]<15:0>;
```

## C2.4.209 STM, STMIA, STMEA

Store Multiple. Store Multiple stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

### T1

*Armv8-M*




#### T1 variant

```
STM{IA}{<c>}{<q>} <Rn>!, <registers>
  // Preferred syntax
STMEA{<c>}{<q>} <Rn>!, <registers>
  // Alternate syntax, Empty Ascending stack
```

### Decode for this encoding

```
1  n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
2  if BitCount(registers) < 1 then UNPREDICTABLE;
```

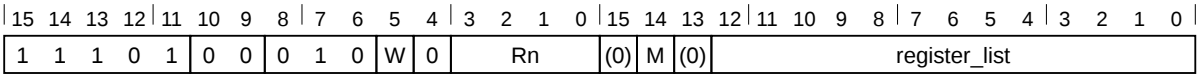
#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
STM{IA}{<c>}.W <Rn>{!}, <registers>
  // Preferred syntax
  // if <Rn>, '!' and <registers> can be represented in T1
STMEA{<c>}.W <Rn>{!}, <registers>
  // Alternate syntax
  // Empty Ascending stack
  // if <Rn>, '!' and <registers> can be represented in T1
STM{IA}{<c>}{<q>} <Rn>{!}, <registers>
  // Preferred syntax
STMEA{<c>}{<q>} <Rn>{!}, <registers>
  // Alternate syntax, Empty Ascending stack
```

## Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
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;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| `IA` | Is an optional suffix for the Increment After form. |
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `!` | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| `<registers>` | For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field. If the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.<br>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0. |

## Operation for all encodings

```
if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    address    = R[n];
    endAddress = R[n] + 4*BitCount(registers);

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback && registers<n> == '0' then
        violatesLimit = ViolatesSPLim(LookUpSP(), endAddress);
    else
        violatesLimit = FALSE;

    for i = 0 to 14
        // Memory operation only performed if limit not violated
        if registers<i> == '1' && !violatesLimit 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 the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = endAddress;
```

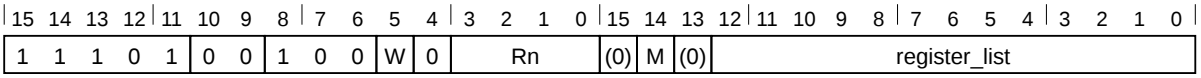
## C2.4.210 STMDB, STMFD

Store Multiple Decrement Before (Full Descending). Store Multiple Decrement Before stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

This instruction is used by the alias PUSH (multiple registers).

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers>
  // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers>
  // Alternate syntax, Full Descending stack
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  registers = '0':M:'0':register_list;  wback = (W == '1');
3  if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
4  if wback && registers<n> == '1' then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && registers<n> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored for the base register is UNKNOWN.

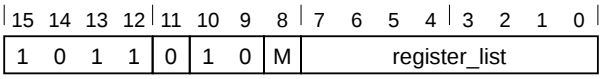
#### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

## T2

*Armv8-M*




### T2 variant

`STMDB{<c>}{<q>} SP!, <registers>`

## Decode for this encoding

```
n = 13; wback = TRUE;
registers = '0':M:'000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15. If the instruction specifies writeback, the modification to the base address on writeback might differ from the number of registers stored.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `!` | The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0. |
| `<registers>` | For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.<br>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    applyLimit = n == 13 && wback;

    for i = 0 to 14
        // If R[n] is the SP, memory operation only performed if limit not violated
        if registers<i> == '1' && !(applyLimit && ViolatesSPLim(LookUpSP(), address)) then
            MemA[address,4] = R[i];
            address = address + 4;

    // If R[n] is the SP, stack pointer update will raise a fault if limit violated
    if wback then RSPCheck[n] = R[n] - 4*BitCount(registers);
```

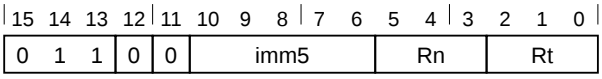
## C2.4.211 STR (immediate)

Store Register (immediate). Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

This instruction is used by the alias PUSH (single register).

### T1

*Armv8-M*




#### T1 variant

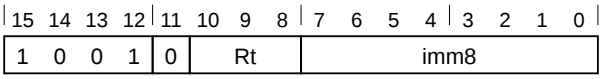
```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M*




#### T2 variant

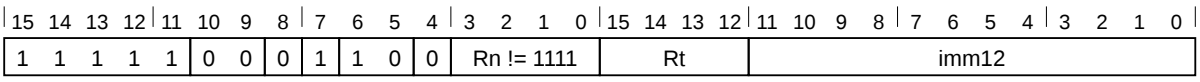
```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

### Decode for this encoding

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

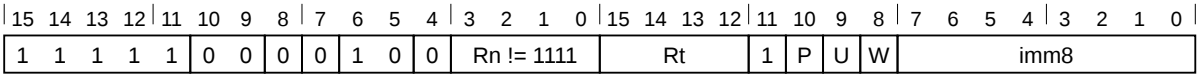
```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1 or T2
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

### T4

*Armv8-M Main Extension only*




**Offset variant**

**Applies when `P == 1 && U == 0 && W == 0`.**

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
STR{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
STR{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
1  if P == '1' && U == '1' && W == '0' then SEE STRT;
2  if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
3  if !HaveMainExt() then UNDEFINED;
4  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
5  index = (P == '1');  add = (U == '1');  wback = (W == '1');
6  if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

## Alias conditions

| Alias | preferred when |
|---|---|
| PUSH (single register) | `Rn == '1101' && U == '0' && imm8 == '00000100'` |

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U == 0`<br>+ when `U == 1` |
| `+` | Specifies the offset is added to the base register. |

<imm> For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.
For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 4 in the range 0 to 124, defaulting to 0 and encoded in the "imm5" field as <imm>/4.
For encoding T2: is the optional positive unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.
For encoding T3: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.
For encoding T4: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation for all encodings

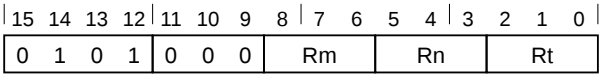
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        MemU[address,4] = R[t];

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

## C2.4.212 STR (register)

Store Register (register). Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

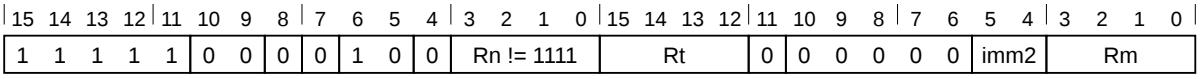
```
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() 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 symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
if ConditionPassed() then
```

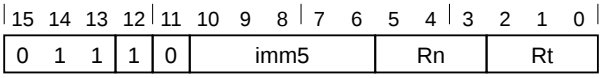
```
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,4] = R[t];
```

## C2.4.213 STRB (immediate)

Store Register Byte (immediate). Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M*




#### T1 variant

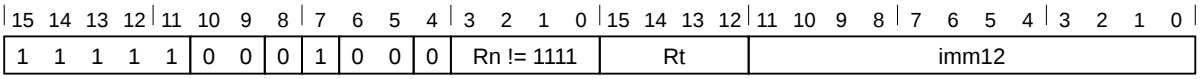
```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5, 32);
2  index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

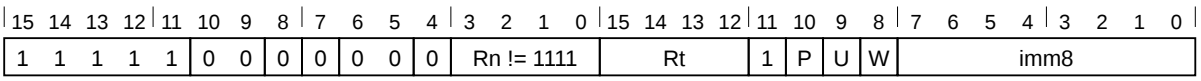
```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then UNDEFINED;
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
4  index = TRUE;  add = TRUE;  wback = FALSE;
5  if t IN {13,15} then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `P == 1 && U == 0 && W == 0`.**

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
STRB{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
STRB{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
if P == '1' && U == '1' && W == '0' then SEE STRBT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
if !HaveMainExt() 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;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>`-` when `U = 0`<br>`+` when `U = 1` |
| `+` | Specifies the offset is added to the base register. |
| `<imm>` | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
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];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
```

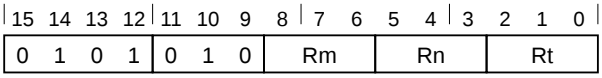
```
13          MemU[address,1] = R[t]<7:0>;
14
15      // If the stack pointer is being updated a fault will be raised if
16      // the limit is violated
17      if wback then RSPCheck[n] = offset_addr;
```

## C2.4.214 STRB (register)

Store Register Byte (register). Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

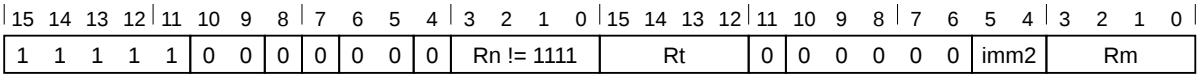
```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() 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 symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
if ConditionPassed() then
```

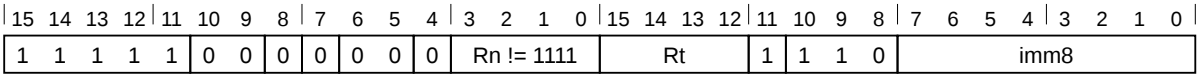
```
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,1] = R[t]<7:0>;
```

## C2.4.215 STRBT

Store Register Byte Unprivileged. Store Register Byte Unprivileged calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. When privileged software uses an STRBT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then UNDEFINED;
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

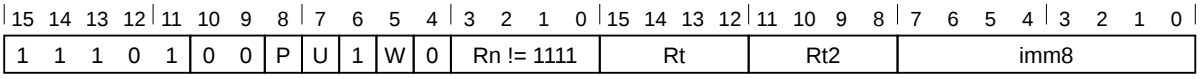
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      MemU_unpriv[address,1] = R[t]<7:0>;
```

## C2.4.216 STRD (immediate)

Store Register Dual (immediate). Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M Main Extension only*




**Offset variant**

**Applies when `P == 1 && W == 0`.**

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

**Post-indexed variant**

**Applies when `P == 0 && W == 1`.**

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

### Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
4  index = (P == '1');  add = (U == '1');  wback = (W == '1');
5  if wback && (n == t || n == t2) then UNPREDICTABLE;
6  if n == 15 || t IN {13,15} || t2 IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rt>` | Is the first general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rt2>` | Is the second general-purpose register to be transferred, encoded in the "Rt2" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `+/-` | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>`-` when `U = 0`<br>`+` when `U = 1` |

<imm> For the offset variant: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0 and encoded in the "imm8" field as <imm>/4.
For the post-indexed and pre-indexed variant: is the unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm>/4.

## Operation for all encodings

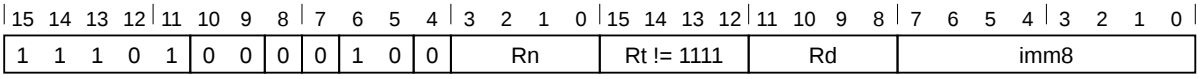
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        MemA[address,4] = R[t];
        MemA[address+4,4] = R[t2];

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = offset_addr;
```

## C2.4.217 STREX

Store Register Exclusive. Store Register Exclusive calculates an address from a base register value and an immediate offset, and stores a word from a register to memory if the executing PE has exclusive access to the memory addressed.

### T1

*Armv8-M*




#### T1 variant

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
2  if t == 15 then SEE "TT";
3  if d IN {13,15} || t == 13 || n == 15 then UNPREDICTABLE;
4  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>`1` If the operation fails to update memory.<br>`0` If the operation updates memory. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |
| `<imm>` | The immediate offset added to the value of `<Rn>` to calculate the address. `<imm>` can be omitted, meaning an offset of 0. Values are multiples of 4 in the range 0-1020. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
```

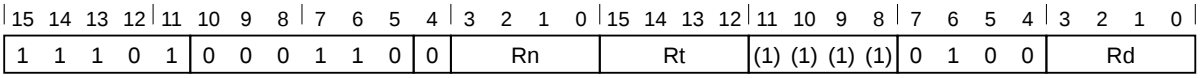
```
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

## C2.4.218 STREXB

Store Register Exclusive Byte. Store Register Exclusive Byte derives an address from a base register value, and stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed.

### T1

*Armv8-M*




#### T1 variant

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
2  if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>`1` If the operation fails to update memory.<br>`0` If the operation updates memory. |
| `<Rt>` | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| `<Rn>` | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      if ExclusiveMonitorsPass(address,1) then
5          MemA[address,1] = R[t]<7:0>;
6          R[d] = ZeroExtend('0');
```

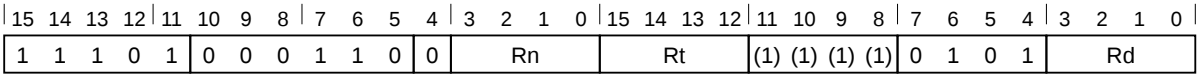
```
7       else
8           R[d] = ZeroExtend('1');
```

## C2.4.219 STREXH

Store Register Exclusive Halfword. Store Register Exclusive Halfword derives an address from a base register value, and stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed.

### T1

*Armv8-M*




#### T1 variant

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

### Decode for this encoding

```
1  d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
2  if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
3  if d == n || d == t then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `d == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `d == n`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction performs the store to an UNKNOWN address.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:<br>1 If the operation fails to update memory.<br>0 If the operation updates memory. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n];
4      if ExclusiveMonitorsPass(address,2) then
5          MemA[address,2] = R[t]<15:0>;
```

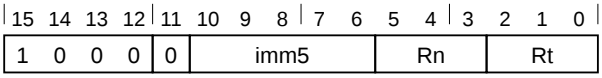
```
6           R[d] = ZeroExtend('0');
7       else
8           R[d] = ZeroExtend('1');
```

## C2.4.220 STRH (immediate)

Store Register Halfword (immediate). Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

### T1

*Armv8-M*




#### T1 variant

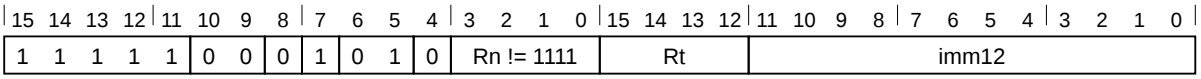
```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
2  index = TRUE;  add = TRUE;  wback = FALSE;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

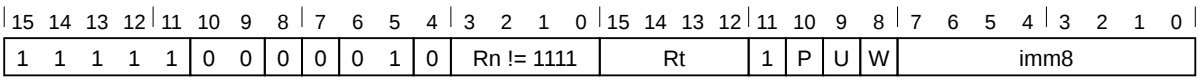
```
STRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}]
  // <Rt>, <Rn>, <imm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then UNDEFINED;
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);
4  index = TRUE;  add = TRUE;  wback = FALSE;
5  if t IN {13,15} then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### Offset variant

**Applies when `P == 1 && U == 0 && W == 0`.**

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

#### Post-indexed variant

**Applies when `P == 0 && W == 1`.**

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #{+/-}<imm>
```

**Pre-indexed variant**

**Applies when `P == 1 && W == 1`.**

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #{+/-}<imm>]!
```

## Decode for this encoding

```
if P == '1' && U == '1' && W == '0' then SEE STRHT;
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
if !HaveMainExt() 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;
```

### CONSTRAINED UNPREDICTABLE behavior

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The store instruction executes but the value stored is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when `U == 0`<br>+ when `U == 1` |
| + | Specifies the offset is added to the base register. |
| <imm> | For the post-indexed or pre-indexed variant: is an 8-bit unsigned immediate byte offset, in the range 0 to 255, encoded in the "imm8" field.<br>For encoding T1: is the optional positive unsigned immediate byte offset, a multiple of 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.<br>For encoding T2: is an optional 12-bit unsigned immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.<br>For encoding T3: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

## Operation for all encodings

```
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];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
```

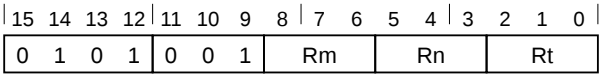
```
13          MemU[address,2] = R[t]<15:0>;
14
15      // If the stack pointer is being updated a fault will be raised if
16      // the limit is violated
17      if wback then RSPCheck[n] = offset_addr;
```

## C2.4.221 STRH (register)

Store Register Halfword (register). Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

### T1

*Armv8-M*




#### T1 variant

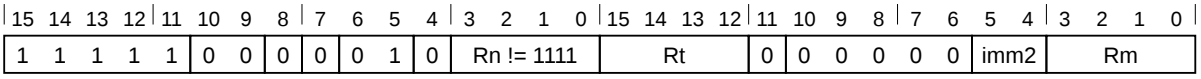
```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
1  t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
2  index = TRUE;  add = TRUE;  wback = FALSE;
3  (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>]
  // <Rt>, <Rn>, <Rm> can be represented in T1
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then UNDEFINED;
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
4  index = TRUE;  add = TRUE;  wback = FALSE;
5  (shift_t, shift_n) = (SRType_LSL, UInt(imm2));
6  if t IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the index register is added to the base register. |
| <Rm> | Is the general-purpose index register, encoded in the "Rm" field. |
| <imm> | If present, the size of the left shift to apply to the value from <Rm>, in the range 1-3. <imm> is encoded in imm2. If absent, no shift is specified and imm2 is encoded as 0b00. |

### Operation for all encodings

```
1  if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,2] = R[t]<15:0>;
```

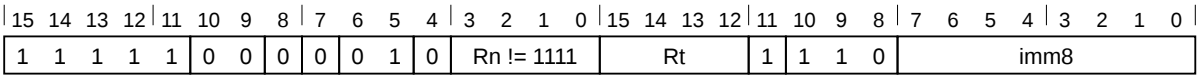
## C2.4.222 STRHT

Store Register Halfword Unprivileged. Store Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory.

When privileged software uses an STRHT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
STRHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1  if Rn == '1111' then UNDEFINED;
2  if !HaveMainExt() then UNDEFINED;
3  t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4  register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5  if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      address = R[n] + imm32;
4      MemU_unpriv[address,2] = R[t]<15:0>;
```

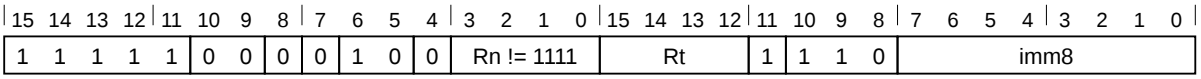
## C2.4.223 STRT

Store Register Unprivileged. Store Register Unprivileged calculates an address from a base register value and an immediate offset, and stores a word from a register to memory.

When privileged software uses an STRT instruction, the memory access is restricted as if the software was unprivileged.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

### Decode for this encoding

```
1 if Rn == '1111' then UNDEFINED;
2 if !HaveMainExt() then UNDEFINED;
3 t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
4 register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
5 if t IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose register to be transferred, encoded in the "Rt" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| + | Specifies the offset is added to the base register. |
| <imm> | Is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field. |

### Operation for all encodings

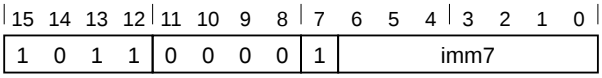
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     address = R[n] + imm32;
4     data = R[t];
5     MemU_unpriv[address,4] = data;
```

## C2.4.224 SUB (SP minus immediate)

Subtract from SP (immediate). Subtract (SP minus immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

### T1

*Armv8-M*




#### T1 variant

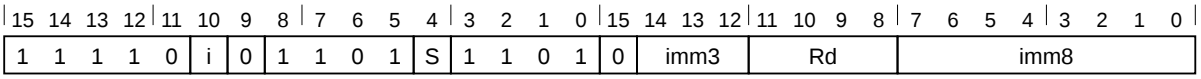
```
SUB{<c>}{<q>} {SP,} SP, #<imm7>
```

### Decode for this encoding

```
1  d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

### T2

*Armv8-M Main Extension only*




#### SUB variant

**Applies when `S == 0`.**

```
SUB{<c>}.W {<Rd>,} SP, #<const>
  // <Rd>, <const> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} SP, #<const>
```

#### SUBS variant

**Applies when `S == 1 && Rd != 1111`.**

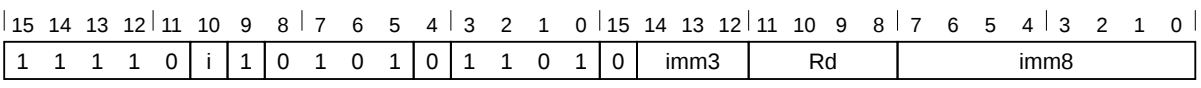
```
SUBS{<c>}{<q>} {<Rd>,} SP, #<const>
```

### Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
4  if d == 15 && S == '0' then UNPREDICTABLE;
```

### T3

*Armv8-M Main Extension only*




#### T3 variant

```
SUB{<c>}{<q>} {<Rd>,} SP, #<imm12>
  // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} SP, #<imm12>
  // <imm12> can be represented in T1, T2, or T3
```

### Decode for this encoding

```
1 if !HaveMainExt() then UNDEFINED;
2 d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
3 if d == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <imm7> | Is an unsigned immediate, a multiple of 4 in the range 0 to 508, encoded in the "imm7" field as <imm7>/4. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

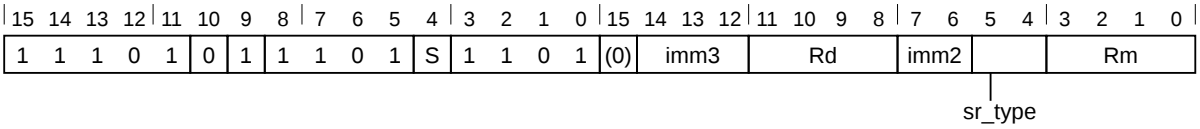
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
4     RSPCheck[d] = result;
5     if setflags then
6         APSR.N = result<31>;
7         APSR.Z = IsZeroBit(result);
8         APSR.C = carry;
9         APSR.V = overflow;
```

## C2.4.225 SUB (SP minus register)

Subtract from SP (register). Subtract (SP minus register) subtracts an optionally-shifted register value from the SP value, and writes the result to the destination register.

### T1

*Armv8-M Main Extension only*




**SUB, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

**SUB, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
SUB{<c>}.W {<Rd>,} SP, <Rm>
  // <Rd>, <Rm> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

**SUBS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11`.**

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

**SUBS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111`.**

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "CMP (register)";
2  if !HaveMainExt() then UNDEFINED;
3  d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
4  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
5  if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
6  if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values: |

LSL when sr_type = 00
LSR when sr_type = 01
ASR when sr_type = 10
ROR when sr_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

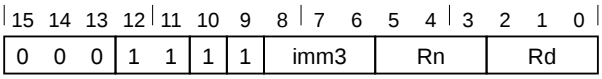
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    RSPCheck[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

## C2.4.226 SUB (immediate)

Subtract (immediate). Subtract (immediate) subtracts an immediate value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

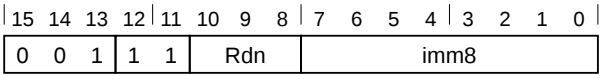
```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3>
  // Inside IT block
SUBS{<q>} <Rd>, <Rn>, #<imm3>
  // Outside IT block
```

### Decode for this encoding

```
1  d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

### T2

*Armv8-M*




**T2 variant**

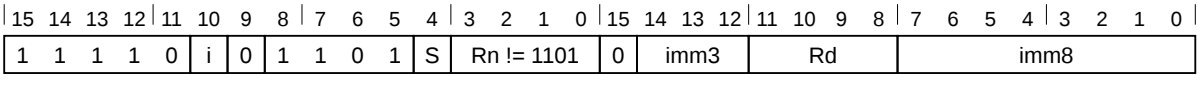
```
SUB<c>{<q>} <Rdn>, #<imm8>
  // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8>
  // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8>
  // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8>
  // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

### Decode for this encoding

```
1  d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

### T3

*Armv8-M Main Extension only*




**SUB variant**

**Applies when `S == 0`.**

```
SUB<c>.W {<Rd>,} <Rn>, #<const>
  // Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

**SUBS variant**

**Applies when `S == 1 && Rd != 1111`.**

```
SUBS.W {<Rd>,} <Rn>, #<const>
  // Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2
SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

## Decode for this encoding

```
1  if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
2  if Rn == '1101' then SEE "SUB (SP minus immediate)"
3  if !HaveMainExt() then UNDEFINED;
4  d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
5  if d == 13 || (d == 15 && S == '0') || n == 15 then UNPREDICTABLE;
```

## T4

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | i | 1 | 0 | 1 | 0 | 1 | 0 | Rn != 11x1 | | | | 0 | imm3 | | | Rd | | | | imm8 | | | | | | | |

**T4 variant**

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, #<imm12>
  // <imm12> cannot be represented in T1, T2, or T3
SUBW{<c>}{<q>} {<Rd>,} <Rn>, #<imm12>
  // <imm12> can be represented in T1, T2, or T3
```

## Decode for this encoding

```
1  if Rn == '1111' then SEE ADR;
2  if Rn == '1101' then SEE "SUB (SP minus immediate)"
3  if !HaveMainExt() then UNDEFINED;
4  d = UInt(Rd);  n = UInt(Rn);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
5  if d IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rdn> | Is the general-purpose source and destination register, encoded in the "Rdn" field. |
| <imm8> | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | For encoding T1: is the general-purpose source register, encoded in the "Rn" field.<br>For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.224 *SUB (SP minus immediate)* on page 850.<br>For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.224 *SUB (SP minus immediate)* on page 850. If the PC is used, see C2.4.8 *ADR* on page 515. |
| <imm3> | Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field. |
| <imm12> | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

```
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;
```

## C2.4.227 SUB (immediate, from PC)

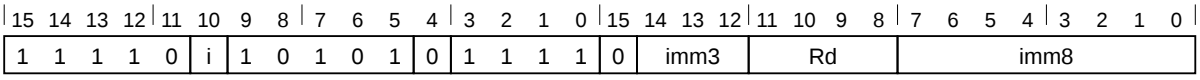
Subtract from PC. Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

### T2

*Armv8-M Main Extension only*




**T2 variant**

`SUB{<c>}{<q>} <Rd>, PC, #<imm12>`

**is equivalent to**

`ADR{<c>}{<q>} <Rd>, <label>`

**and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.**

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<Rd>` | Is the general-purpose destination register, encoded in the "Rd" field. |
| `<label>` | For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the Align `(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020.<br>For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the Align `(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095. |
| `<imm12>` | Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field. |

### Operation for all encodings

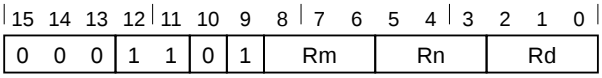
The description of ADR gives the operational pseudocode for this instruction.

## C2.4.228 SUB (register)

Subtract (register). Subtract (register) subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

### T1

*Armv8-M*




**T1 variant**

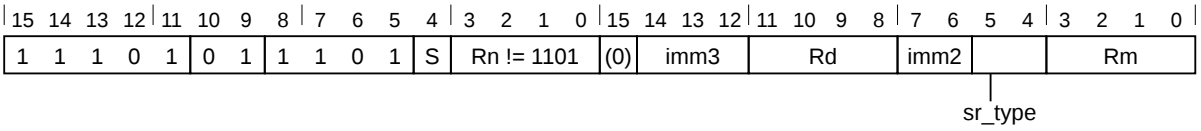
```
SUB<c>{<q>} <Rd>, <Rn>, <Rm>
  // Inside IT block
SUBS{<q>} {<Rd>,} <Rn>, <Rm>
  // Outside IT block
```

### Decode for this encoding

```
1 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
2 (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




**SUB, rotate right with extend variant**

**Applies when `S == 0 && imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**SUB, shift or rotate by value variant**

**Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
SUB<c>.W {<Rd>,} <Rn>, <Rm>
  // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

**SUBS, rotate right with extend variant**

**Applies when `S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && sr_type == 11`.**

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

**SUBS, shift or rotate by value variant**

**Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && sr_type == 11) && Rd != 1111`.**

```
SUBS.W {<Rd>,} <Rn>, <Rm>
  // Outside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

## Decode for this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)"
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>. |
| <Rn> | For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.<br>For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see C2.4.225 *SUB (SP minus register)* on page 852. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when sr_type = 00<br>LSR when sr_type = 01<br>ASR when sr_type = 10<br>ROR when sr_type = 11 |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

## Operation for all encodings

```
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;
```

## C2.4.229 SVC

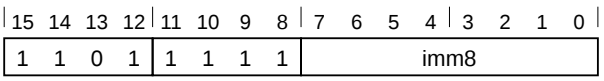
Supervisor Call. The Supervisor Call instruction generates a call to a system supervisor.

Use it as a call to an operating system to provide a service.

In older versions of the Arm architecture, `SVC` was called `SWI`, Software Interrupt.

### T1

*Armv8-M*




#### T1 variant

`SVC{<c>}{<q>} {#}<imm>`

### Decode for this encoding

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<imm>` | Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. |

### Operation for all encodings

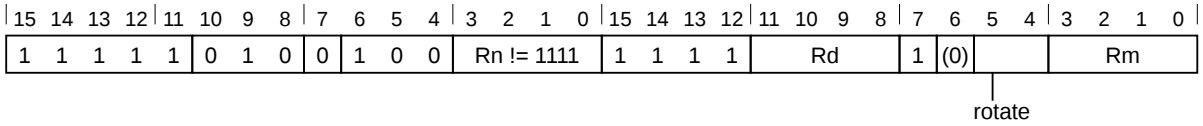
```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

## C2.4.230 SXTAB

Signed Extend and Add Byte. Signed Extend and Add Byte extracts an 8-bit value from a register, sign-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE SXTB;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4  if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- 0 when `rotate = 00`
- 8 when `rotate = 01`
- 16 when `rotate = 10`
- 24 when `rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

### Operation for all encodings

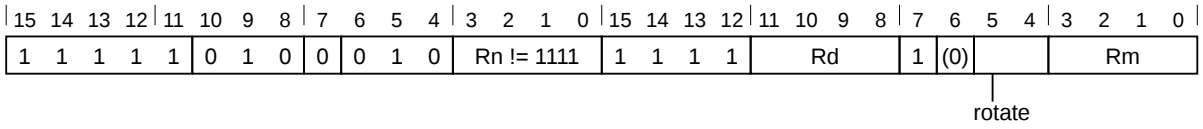
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

## C2.4.231 SXTAB16

Signed Extend and Add Byte 16. Signed Extend and Add Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE SXTB16;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4  if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

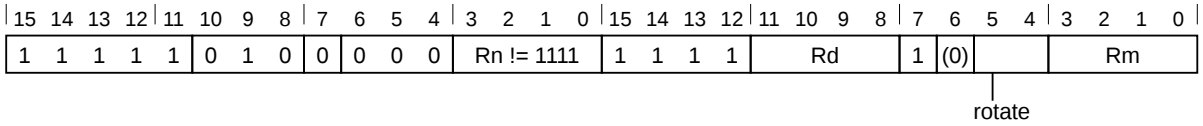
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      bits(32) result;
5      result<15:0>  = R[n]<15:0>  + SignExtend(rotated<7:0>,   16);
6      result<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
7      R[d] = result;
```

## C2.4.232 SXTAH

Signed Extend and Add Halfword. Signed Extend and Add Halfword extracts a 16-bit value from a register, sign-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1 if Rn == '1111' then SEE SXTH;
2 if !HaveDSPExt() then UNDEFINED;
3 d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4 if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

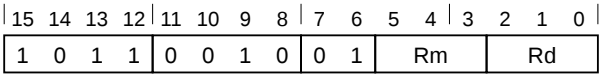
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     rotated = ROR(R[m], rotation);
4     R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

## C2.4.233 SXTB

Signed Extend Byte. Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

*Armv8-M*




#### T1 variant

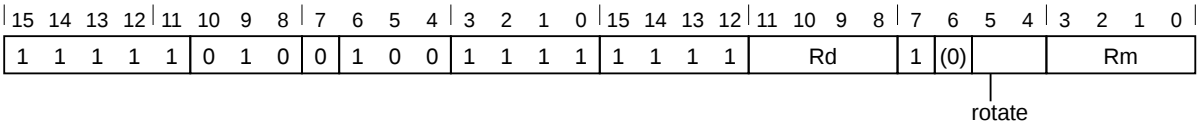
```
SXTB{<c>}{<q>} {<Rd>,} <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
SXTB{<c>}.W {<Rd>,} <Rm>
  // <Rd>, <Rm> can be represented in T1
SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
3  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

## Operation for all encodings

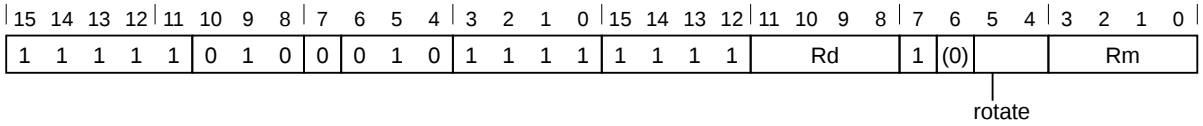
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

## C2.4.234 SXTB16

Signed Extend Byte 16. Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
3  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

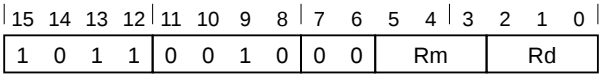
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      bits(32) result;
5      result<15:0>  = SignExtend(rotated<7:0>,   16);
6      result<31:16> = SignExtend(rotated<23:16>, 16);
7      R[d] = result;
```

## C2.4.235 SXTH

Signed Extend Halfword. Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

*Armv8-M*




#### T1 variant

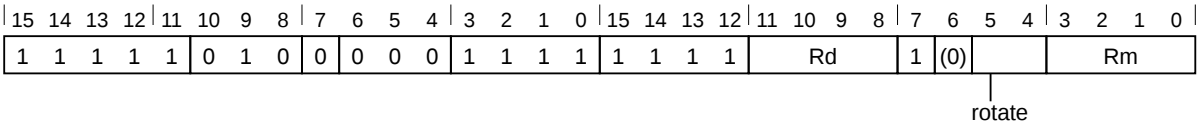
```
SXTH{<c>}{<q>} {<Rd>,} <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
SXTH{<c>}.W {<Rd>,} <Rm>
  // <Rd>, <Rm> can be represented in T1
SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
3  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

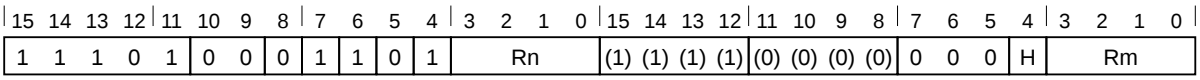
## C2.4.236 TBB, TBH

Table Branch Byte or Halfword. Table Branch Byte causes a PC-relative forward branch using a table of single byte offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the byte returned from the table.

Table Branch Halfword causes a PC-relative forward branch using a table of single halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the halfword returned from the table.

### T1

*Armv8-M Main Extension only*




**Byte variant**

**Applies when `H == 0`.**

```
TBB{<c>}{<q>} [<Rn>, <Rm>]
  // Outside or last in IT block
```

**Halfword variant**

**Applies when `H == 1`.**

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1]
  // Outside or last in IT block
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
3  if n == 13 || m IN {13,15} then UNPREDICTABLE;
4  if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rn> Is the general-purpose base register holding the address of the table of branch lengths, encoded in the "Rn" field. The PC can be used. If it is, the table immediately follows this instruction.

<Rm> For the byte variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a single byte in the table. The offset in the table is the value of the index.
For the halfword variant: is the general-purpose index register, encoded in the "Rm" field. This register contains an integer pointing to a halfword in the table. The offset in the table is twice the value of the index.

### Operation for all encodings

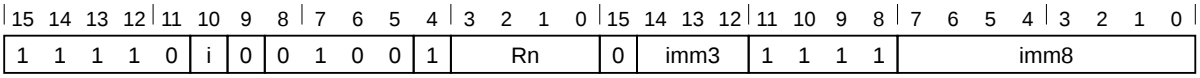
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if is_tbh then
4          halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
5      else
6          halfwords = UInt(MemU[R[n]+R[m], 1]);
7      BranchTo(PC + 2*halfwords);
```

## C2.4.237 TEQ (immediate)

Test Equivalence (immediate). Test Equivalence (immediate) performs an exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4  if n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

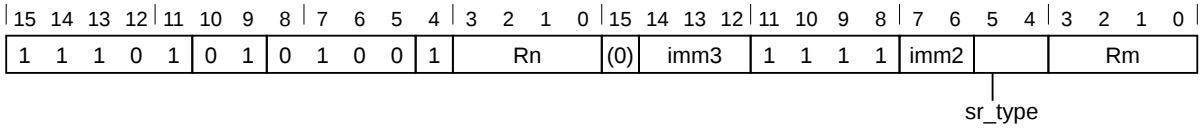
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = R[n] EOR imm32;
4      APSR.N = result<31>;
5      APSR.Z = IsZeroBit(result);
6      APSR.C = carry;
7      // APSR.V unchanged
```

## C2.4.238 TEQ (register)

Test Equivalence (register). Test Equivalence (register) performs an exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M Main Extension only*




**Rotate right with extend variant**

**Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

**Shift or rotate by value variant**

**Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  m = UInt(Rm);
3  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4  if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values:<br>LSL when `sr_type = 00`<br>LSR when `sr_type = 01`<br>ASR when `sr_type = 10`<br>ROR when `sr_type = 11` |
| <amount> | Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32. |

### Operation for all encodings

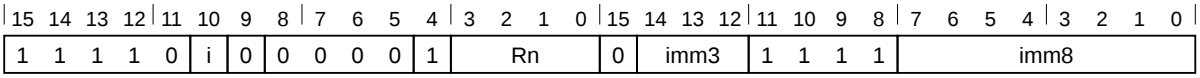
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
4      result = R[n] EOR shifted;
5      APSR.N = result<31>;
6      APSR.Z = IsZeroBit(result);
7      APSR.C = carry;
8      // APSR.V unchanged
```

## C2.4.239 TST (immediate)

Test (immediate). Test (immediate) performs a bitwise AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
TST{<c>}{<q>} <Rn>, #<const>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
4  if n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <const> | Is an immediate value derived from the 12-bit immediate that is encoded in the 'i:imm3:imm8' field. See C1.5 *Modified immediate constants* on page 441 for the range of values. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = R[n] AND imm32;
4      APSR.N = result<31>;
5      APSR.Z = IsZeroBit(result);
6      APSR.C = carry;
7      // APSR.V unchanged
```

## C2.4.240 TST (register)

Test (register). Test (register) performs a bitwise AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

### T1

*Armv8-M*


#### T1 variant

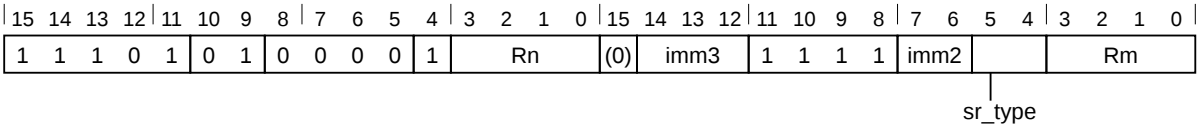
```
TST{<c>}{<q>} <Rn>, <Rm>
```

### Decode for this encoding

```
1  n = UInt(Rn);  m = UInt(Rm);
2  (shift_t, shift_n) = (SRType_LSL, 0);
```

### T2

*Armv8-M Main Extension only*




#### Rotate right with extend variant

**Applies when `imm3 == 000 && imm2 == 00 && sr_type == 11`.**

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

#### Shift or rotate by value variant

**Applies when `!(imm3 == 000 && imm2 == 00 && sr_type == 11)`.**

```
TST{<c>}.W <Rn>, <Rm>
  // <Rn>, <Rm> can be represented in T1
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);  m = UInt(Rm);
3  (shift_t, shift_n) = DecodeImmShift(sr_type, imm3:imm2);
4  if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <shift> | Is the type of shift to be applied to the second source register, encoded in the "sr_type" field. It can have the following values: |

LSL when sr_type = 00
LSR when sr_type = 01
ASR when sr_type = 10
ROR when sr_type = 11

<amount> Is the shift amount, in the range 1 to 31 (when <shift> = LSL or ROR) or 1 to 32 (when <shift> = LSR or ASR) encoded in the "imm3:imm2" field as <amount> modulo 32.

## Operation for all encodings

```
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
```

## C2.4.241 TT, TTT, TTA, TTAT

Test Target (Alternate Domain, Unprivileged). Test Target (TT) queries the Security state and access permissions of a memory location.

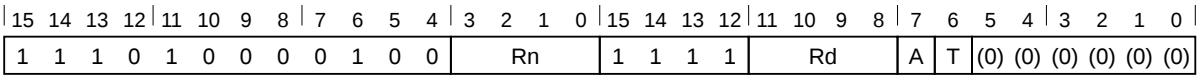
Test Target Unprivileged (TTT) queries the Security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the Security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

These instructions return the Security state and access permissions in the destination register. See TT_RESP for the format of the destination register.

### T1

*Armv8-M*




**TT variant**

**Applies when `A == 0 && T == 0`.**

```
TT{<c>}{<q>} <Rd>, <Rn>
```

**TTA variant**

**Applies when `A == 1 && T == 0`.**

```
TTA{<c>}{<q>} <Rd>, <Rn>
```

**TTAT variant**

**Applies when `A == 1 && T == 1`.**

```
TTAT{<c>}{<q>} <Rd>, <Rn>
```

**TTT variant**

**Applies when `A == 0 && T == 1`.**

```
TTT{<c>}{<q>} <Rd>, <Rn>
```

### Decode for this encoding

```
1 d = UInt(Rd); n = UInt(Rn); alt = (A == '1'); forceunpriv = (T == '1');
2 if d IN {13,15} || n == 15 then UNPREDICTABLE;
3 if alt && !IsSecure() then UNDEFINED;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the destination general-purpose register into which the status result of the target test is written, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |

## Operation for all encodings

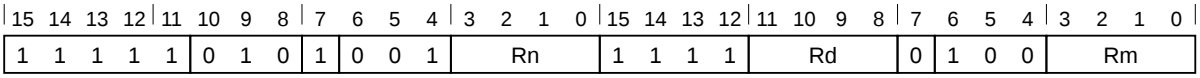
```
if ConditionPassed() then
    EncodingSpecificOperations();
    addr = R[n];
    R[d] = TTResp(addr, alt, forceunpriv);
```

## C2.4.242 UADD16

Unsigned Add 16. Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

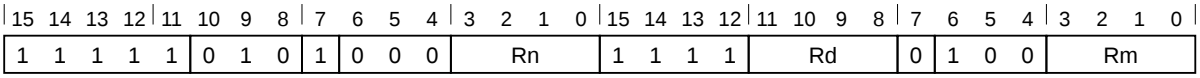
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
4      sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
5      R[d] = sum2<15:0> : sum1<15:0>;
6      APSR.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
7      APSR.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

## C2.4.243 UADD8

Unsigned Add 8. Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

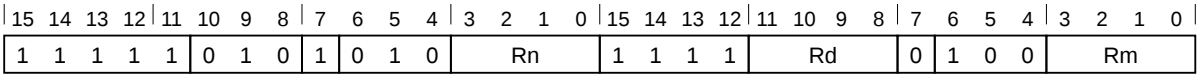
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
4      sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
5      sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
6      sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
7      R[d] = sum4<7:0> : sum3<7:0> : sum2<7:0> : sum1<7:0>;
8      APSR.GE<0>  = if sum1 >= 0x100 then '1' else '0';
9      APSR.GE<1>  = if sum2 >= 0x100 then '1' else '0';
10     APSR.GE<2>  = if sum3 >= 0x100 then '1' else '0';
11     APSR.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

## C2.4.244 UASX

Unsigned Add and Subtract with Exchange. Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

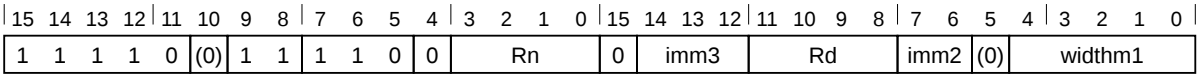
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
4      sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
5      R[d] = sum<15:0> : diff<15:0>;
6      APSR.GE<1:0> = if diff >= 0 then '11' else '00';
7      APSR.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

## C2.4.245 UBFX

Unsigned Bit Field Extract. Unsigned Bit Field Extract extracts any number of adjacent bits at any position from one register, zero extends them to 32 bits, and writes the result to the destination register.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);
3  lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
4  msbit = lsbit + widthminus1;
5  if msbit > 31 then UNPREDICTABLE;
6  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `msbit > 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <lsb> | Is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "imm3:imm2" field. |
| <width> | Is the width of the field, in the range 1 to 32-<lsb>, encoded in the "widthm1" field as <width>-1. |

### Operation for all encodings

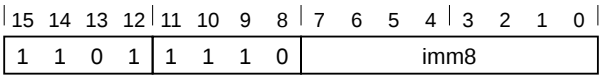
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if msbit <= 31 then
4          R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
5      else
6          R[d] = bits(32) UNKNOWN;
```

## C2.4.246 UDF

Permanently Undefined. Permanently Undefined generates an Undefined Instruction exception.

### T1

*Armv8-M*




#### T1 variant

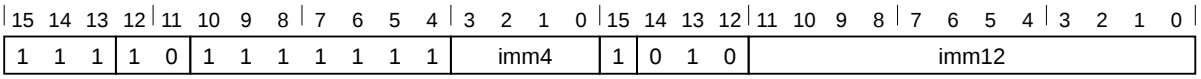
```
UDF{<c>}{<q>} {#}<imm>
```

### Decode for this encoding

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

### T2

*Armv8-M*




#### T2 variant

```
UDF{<c>}.W {#}<imm>
  // <imm> can be represented in T1
UDF{<c>}{<q>} {#}<imm>
```

### Decode for this encoding

```
imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. Arm deprecates using any <c> value other than AL. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <imm> | For encoding T1: is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant.<br>For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant. |

### Operation for all encodings

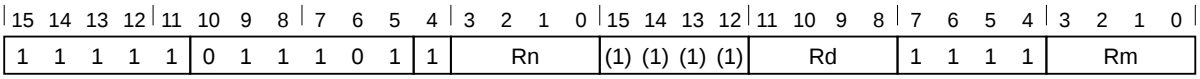
```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

## C2.4.247 UDIV

Unsigned Divide. Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

### T1

*Armv8-M*




#### T1 variant

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
2  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register holding the dividend, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the divisor, encoded in the "Rm" field. |

### Operation for all encodings

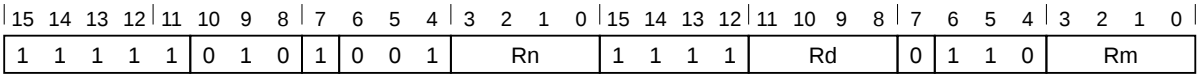
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if UInt(R[m]) == 0 then
4          if IntegerZeroDivideTrappingEnabled() then
5              GenerateIntegerZeroDivide();
6          else
7              result = 0;
8      else
9          result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
10     R[d] = result<31:0>;
```

## C2.4.248 UHADD16

Unsigned Halving Add 16. Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

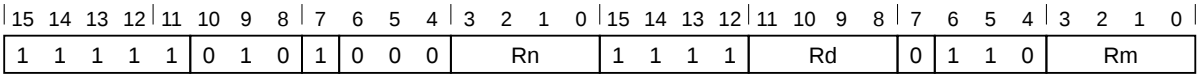
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
4      sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
5      R[d] = sum2<16:1> : sum1<16:1>;
```

## C2.4.249 UHADD8

Unsigned Halving Add 8. Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

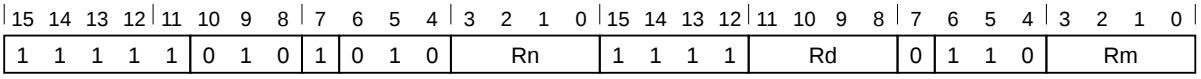
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
4      sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
5      sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
6      sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
7      R[d] = sum4<8:1> : sum3<8:1> : sum2<8:1> : sum1<8:1>;
```

## C2.4.250 UHASX

Unsigned Halving Add and Subtract with Exchange. Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

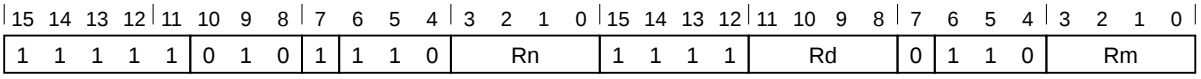
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
4      sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
5      R[d] = sum<16:1> : diff<16:1>;
```

## C2.4.251 UHSAX

Unsigned Halving Subtract and Add with Exchange. Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

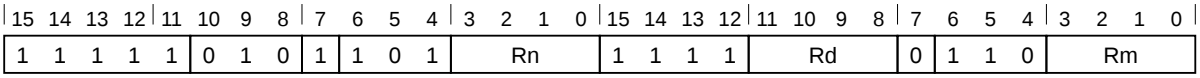
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
4      diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
5      R[d] = diff<16:1> : sum<16:1>;
```

## C2.4.252 UHSUB16

Unsigned Halving Subtract 16. Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

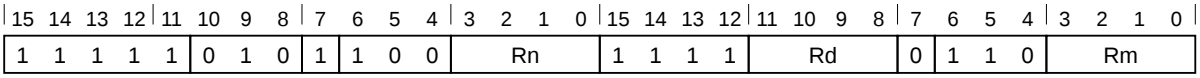
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
4      diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
5      R[d]  = diff2<16:1> : diff1<16:1>;
```

## C2.4.253 UHSUB8

Unsigned Halving Subtract 8. Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

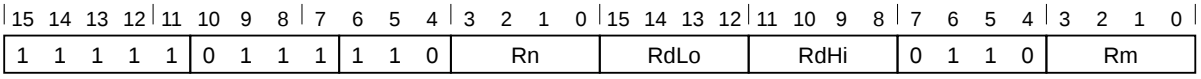
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
4      diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
5      diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
6      diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
7      R[d]  = diff4<8:1> : diff3<8:1> : diff2<8:1> : diff1<8:1>;
```

## C2.4.254 UMAAL

Unsigned Multiply Accumulate Accumulate Long. Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

`UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>`

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<RdLo>` | Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| `<RdHi>` | Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| `<Rn>` | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| `<Rm>` | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation for all encodings

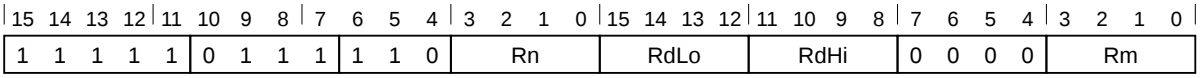
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
4      R[dHi] = result<63:32>;
5      R[dLo] = result<31:0>;
```

## C2.4.255 UMLAL

Unsigned Multiply Accumulate Long. Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

### T1

*Armv8-M Main Extension only*




#### T1 variant

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation for all encodings

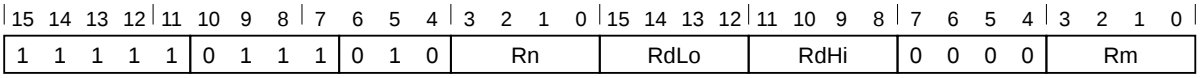
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
4      R[dHi] = result<63:32>;
5      R[dLo] = result<31:0>;
```

## C2.4.256 UMULL

Unsigned Multiply Long. Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

### T1

*Armv8-M Main Extension only*




#### T1 variant

`UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>`

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
3  if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
4  if dHi == dLo then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `dHi == dLo`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdLo> | Is the general-purpose destination register for the lower 32 bits of the result, encoded in the "RdLo" field. |
| <RdHi> | Is the general-purpose destination register for the upper 32 bits of the result, encoded in the "RdHi" field. |
| <Rn> | Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field. |

### Operation for all encodings

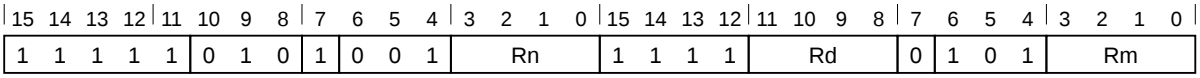
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      result = UInt(R[n]) * UInt(R[m]);
4      R[dHi] = result<63:32>;
5      R[dLo] = result<31:0>;
```

## C2.4.257 UQADD16

Unsigned Saturating Add 16. Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 to $2^{16}$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

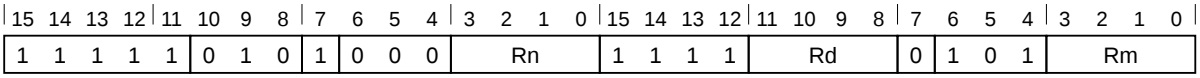
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
4      sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
5      bits(32) result;
6      result<15:0>  = UnsignedSat(sum1, 16);
7      result<31:16> = UnsignedSat(sum2, 16);
8      R[d] = result;
```

## C2.4.258 UQADD8

Unsigned Saturating Add 8. Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 to $2^8$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

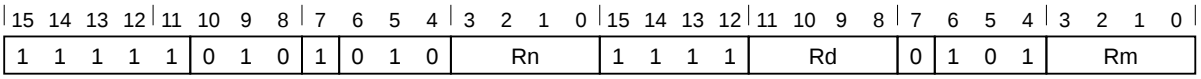
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
4      sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
5      sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
6      sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
7      bits(32) result;
8      result<7:0>   = UnsignedSat(sum1, 8);
9      result<15:8>  = UnsignedSat(sum2, 8);
10     result<23:16> = UnsignedSat(sum3, 8);
11     result<31:24> = UnsignedSat(sum4, 8);
12     R[d] = result;
```

## C2.4.259 UQASX

Unsigned Saturating Add and Subtract with Exchange. Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 to $2^{16}$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

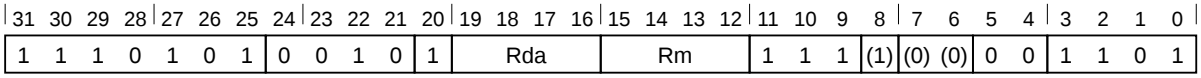
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
4      sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
5      bits(32) result;
6      result<15:0>  = UnsignedSat(diff, 16);
7      result<31:16> = UnsignedSat(sum,  16);
8      R[d] = result;
```

## C2.4.260 UQRSHL (register)

Unsigned Saturating Rounding Shift Left. Unsigned saturating rounding shift left by 0 to 32 bits of a 32 bit value stored in a general-purpose register. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: UQRSHL variant

```
UQRSHL<c> Rda, Rm
```

### Decode for this encoding

```
if !HaveMainExt()            then UNDEFINED;
if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
if !HaveMve()                then UNDEFINED;
da = UInt(Rda);
m  = UInt(Rm);
if Rda == '11x1' || Rm == '11x1' || Rm == Rda then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

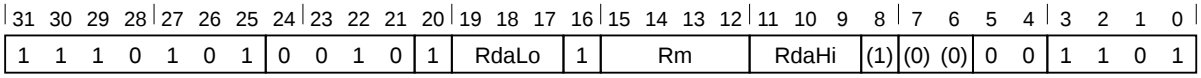
```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount = SInt(R[m]<7:0>);
    op1    = UInt(R[da]);
    op1    = op1 + (1 << (- 1 - amount));
    (result, sat) = UnsignedSatQ((op1 << amount), 32);
    if sat then APSR.Q = '1';
    R[da] = result<31:0>;
```

## C2.4.261 UQRSHLL (register)

Unsigned Saturating Rounding Shift Left Long. Unsigned saturating rounding shift left by 0 to 64 bits of a 64 bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

### T1

*Armv8.1-M MVE*




#### T1: UQRSHLL variant

`UQRSHLL<c> RdaLo, RdaHi, Rm`

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "UQRSHL (register)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  m   = UInt(Rm);
8  if RdaHi == '110' || Rm == '11x1' || Rm == RdaHi:'1' then CONSTRAINED_UNPREDICTABLE;
9  if Rm    == RdaLo:'0' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <Rm> | General-purpose source register holding a shift amount in its bottom 8 bits. |

### Operation for all encodings

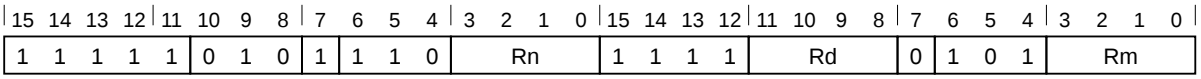
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      amount = SInt(R[m]<7:0>);
5      op1    = UInt(R[dah]:R[dal]);
6      op1    = op1 + (1 << (- 1 - amount));
7      (result, sat) = UnsignedSatQ((op1 << amount), 64);
8      if sat then APSR.Q = '1';
9      R[dah] = result<63:32>;
10     R[dal] = result<31:0>;
```

## C2.4.262 UQSAX

Unsigned Saturating Subtract and Add with Exchange. Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 to $2^{16}$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

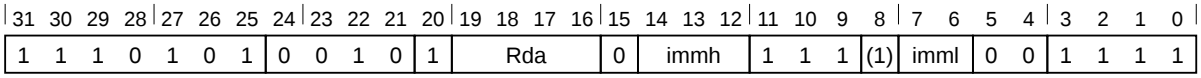
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
4      diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
5      bits(32) result;
6      result<15:0>  = UnsignedSat(sum,  16);
7      result<31:16> = UnsignedSat(diff, 16);
8      R[d] = result;
```

## C2.4.263 UQSHL (immediate)

Unsigned Saturating Shift Left. Unsigned saturating shift left by 1 to 32 bits of a 32 bit value stored in a general-purpose register.

### T1

*Armv8.1-M MVE*




#### T1: UQSHL variant

```
UQSHL<c> Rda, #<imm>
```

### Decode for this encoding

```
1  if !HaveMainExt()           then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3  if !HaveMve()               then UNDEFINED;
4  da = UInt(Rda);
5  (-, amount) = DecodeImmShift('10', immh:imml);
6  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

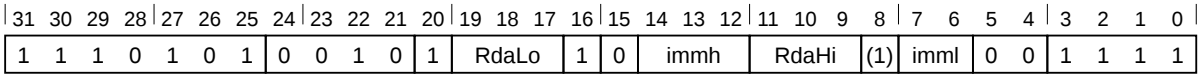
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1 = UInt(R[da]);
5      (result, sat) = UnsignedSatQ((op1 << amount), 32);
6      if sat then APSR.Q = '1';
7      R[da] = result<31:0>;
```

## C2.4.264 UQSHLL (immediate)

Unsigned Saturating Shift Left Long. Unsigned saturating shift left by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: UQSHLL variant

```
UQSHLL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "UQSHL (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

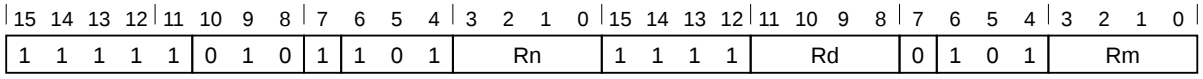
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1 = UInt(R[dah]:R[dal]);
5      (result, sat) = UnsignedSatQ((op1 << amount), 64);
6      if sat then APSR.Q = '1';
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

## C2.4.265 UQSUB16

Unsigned Saturating Subtract 16. Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 to $2^{16}$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

`UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>`

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

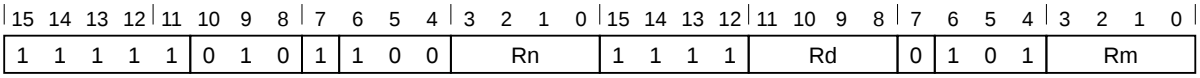
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
4      diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
5      bits(32) result;
6      result<15:0>  = UnsignedSat(diff1, 16);
7      result<31:16> = UnsignedSat(diff2, 16);
8      R[d] = result;
```

## C2.4.266 UQSUB8

Unsigned Saturating Subtract 8. Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 to $2^8$-1, and writes the results to the destination register.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

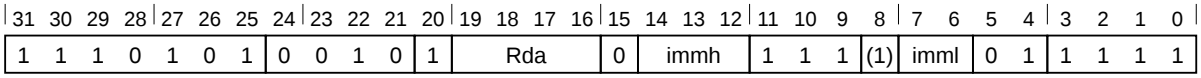
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
4      diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
5      diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
6      diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
7      bits(32) result;
8      result<7:0>   = UnsignedSat(diff1, 8);
9      result<15:8>  = UnsignedSat(diff2, 8);
10     result<23:16> = UnsignedSat(diff3, 8);
11     result<31:24> = UnsignedSat(diff4, 8);
12     R[d] = result;
```

## C2.4.267 URSHR (immediate)

Unsigned Rounding Shift Right. Unsigned rounding shift right by 1 to 32 bits of a 32 bit value stored in a general-purpose register.

### T1

*Armv8.1-M MVE*




#### T1: URSHR variant

```
URSHR<c> Rda, #<imm>
```

### Decode for this encoding

```
1  if !HaveMainExt()           then UNDEFINED;
2  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
3  if !HaveMve()               then UNDEFINED;
4  da = UInt(Rda);
5  (-, amount) = DecodeImmShift('10', immh:imml);
6  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register, containing the value to be shifted. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

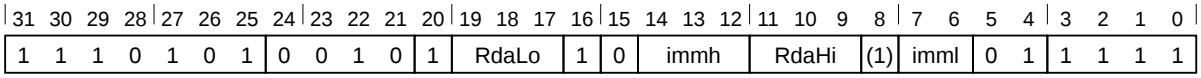
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = UInt(R[da]);
5      op1    = op1 + (1 << (amount - 1));
6      result = (op1 >> amount)<31:0>;
7      R[da]  = result<31:0>;
```

## C2.4.268 URSHRL (immediate)

Unsigned Rounding Shift Right Long. Unsigned rounding shift right by 1 to 32 bits of a 64 bit value stored in two general-purpose registers.

### T1

*Armv8.1-M MVE*




#### T1: URSHRL variant

```
URSHRL<c> RdaLo, RdaHi, #<imm>
```

### Decode for this encoding

```
1  if RdaHi == '111'           then SEE "URSHR (immediate)";
2  if !HaveMainExt()           then UNDEFINED;
3  if !HasArchVersion(Armv8p1) then CONSTRAINED_UNPREDICTABLE;
4  if !HaveMve()               then UNDEFINED;
5  dah = UInt(RdaHi:'1');
6  dal = UInt(RdaLo:'0');
7  (-, amount) = DecodeImmShift('10', immh:imml);
8  if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination, containing the value to be shifted. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64 bit source and destination, containing the value to be shifted. This must be an odd numbered register. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

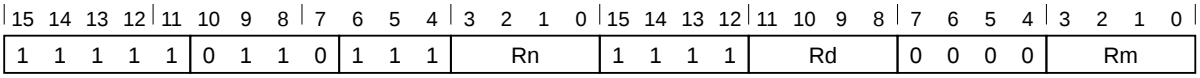
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3
4      op1    = UInt(R[dah]:R[dal]);
5      op1    = op1 + (1 << (amount - 1));
6      result = (op1 >> amount)<63:0>;
7      R[dah] = result<63:32>;
8      R[dal] = result<31:0>;
```

## C2.4.269 USAD8

Unsigned Sum of Absolute Differences. Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

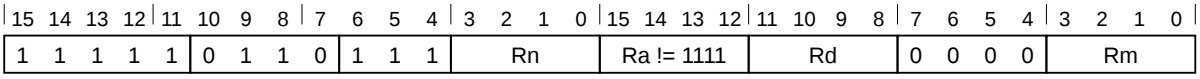
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
4      absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
5      absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
6      absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
7      result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
8      R[d] = result<31:0>;
```

## C2.4.270 USADA8

Unsigned Sum of Absolute Differences and Accumulate. Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

### Decode for this encoding

```
1  if Ra == '1111' then SEE USAD8;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
4  if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <Ra> | Is the third general-purpose source register holding the addend, encoded in the "Ra" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
4      absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
5      absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
6      absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
7      result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
8      R[d] = result<31:0>;
```

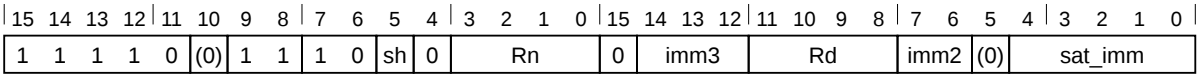
## C2.4.271 USAT

Unsigned Saturate. Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

The Q flag is set to 1 if the operation saturates.

### T1

*Armv8-M Main Extension only*




**Arithmetic shift right variant**

**Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.**

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

**Logical shift left variant**

**Applies when `sh == 0`.**

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

### Decode for this encoding

```
if sh == '1' && (imm3:imm2) == '00000' then
    if HaveDSPExt() then
        SEE USAT16;
    else
        UNDEFINED;
if !HaveMainExt() 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 symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 0 to 31, encoded in the "sat_imm" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |
| <amount> | For the arithmetic shift right variant: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.<br>For the logical shift left variant: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. |

### Operation for all encodings

```
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';
```

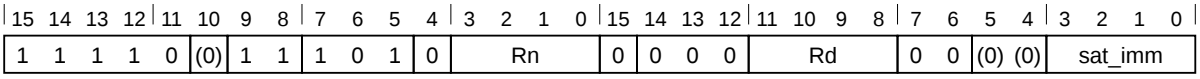
## C2.4.272 USAT16

Unsigned Saturate 16. Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

The Q flag is set to 1 if the operation saturates.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
3  if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <imm> | Is the bit position for saturation, in the range 0 to 15, encoded in the "sat_imm" field. |
| <Rn> | Is the general-purpose source register, encoded in the "Rn" field. |

### Operation for all encodings

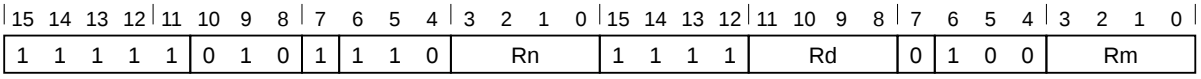
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
4      (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
5      bits(32) result;
6      result<15:0>  = ZeroExtend(result1, 16);
7      result<31:16> = ZeroExtend(result2, 16);
8      R[d] = result;
9      if sat1 || sat2 then
10         APSR.Q = '1';
```

## C2.4.273 USAX

Unsigned Subtract and Add with Exchange. Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

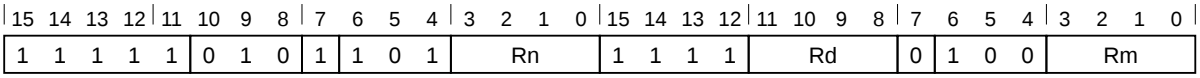
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
4      diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
5      R[d] = diff<15:0> : sum<15:0>;
6      APSR.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
7      APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

## C2.4.274 USUB16

Unsigned Subtract 16. Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

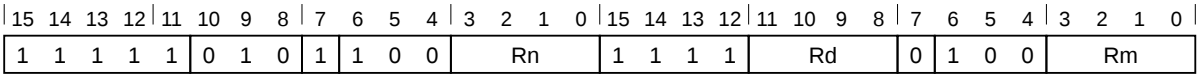
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
4      diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
5      R[d]  = diff2<15:0> : diff1<15:0>;
6      APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
7      APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

## C2.4.275 USUB8

Unsigned Subtract 8. Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
3  if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |

### Operation for all encodings

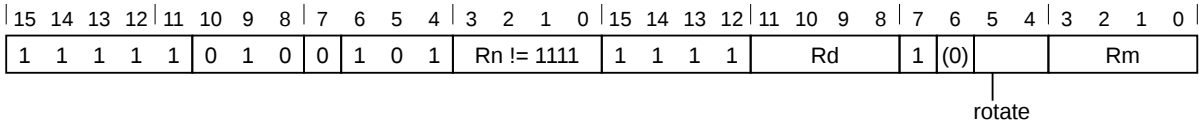
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
4      diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
5      diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
6      diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
7      R[d]  = diff4<7:0> : diff3<7:0> : diff2<7:0> : diff1<7:0>;
8      APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
9      APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
10     APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
11     APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

## C2.4.276 UXTAB

Unsigned Extend and Add Byte. Unsigned Extend and Add Byte extracts an 8-bit value from a register, zero-extends it to 32 bits, adds the result to the value in another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE UXTB;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4  if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rn> | Is the first general-purpose source register, encoded in the "Rn" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

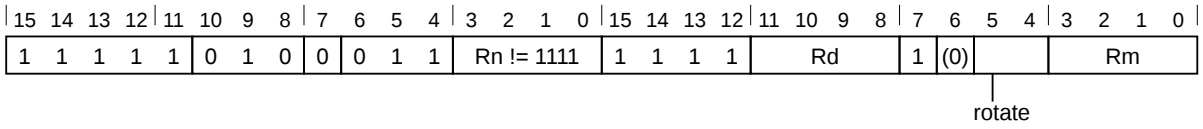
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

## C2.4.277 UXTAB16

Unsigned Extend and Add Byte 16. Unsigned Extend and Add Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, adds the results to two 16-bit values from another register, and writes the final results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE UXTB16;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4  if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- 0 when `rotate = 00`
- 8 when `rotate = 01`
- 16 when `rotate = 10`
- 24 when `rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

### Operation for all encodings

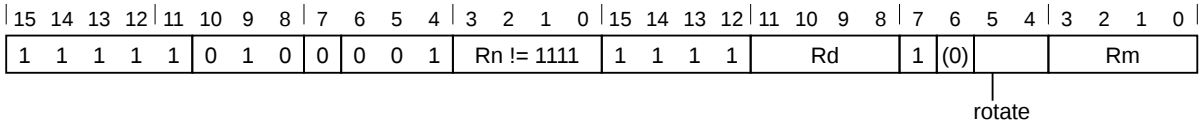
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      bits(32) result;
5      result<15:0>  = R[n]<15:0>  + ZeroExtend(rotated<7:0>,   16);
6      result<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
7      R[d] = result;
```

## C2.4.278 UXTAH

Unsigned Extend and Add Halfword. Unsigned Extend and Add Halfword extracts a 16-bit value from a register, zero-extends it to 32 bits, adds the result to a value from another register, and writes the final result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE UXTH;
2  if !HaveDSPExt() then UNDEFINED;
3  d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
4  if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register, encoded in the "Rm" field.

<amount> Is the rotate amount, encoded in the "rotate" field. It can have the following values:

- 0 when `rotate = 00`
- 8 when `rotate = 01`
- 16 when `rotate = 10`
- 24 when `rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

### Operation for all encodings

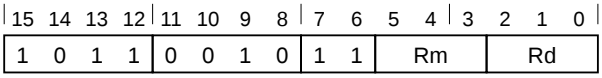
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

## C2.4.279 UXTB

Unsigned Extend Byte. Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

### T1

*Armv8-M*




#### T1 variant

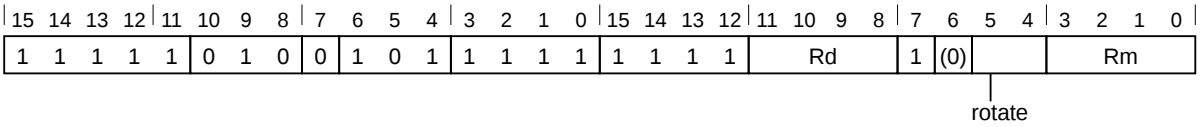
```
UXTB{<c>}{<q>} {<Rd>,} <Rm>
```

### Decode for this encoding

```
1  d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
UXTB{<c>}.W {<Rd>,} <Rm>
  // <Rd>, <Rm> can be represented in T1
UXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
3  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

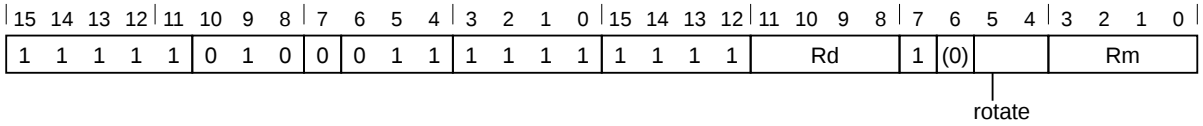
```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

## C2.4.280 UXTB16

Unsigned Extend Byte 16. Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

### T1

*Armv8-M DSP Extension only*




#### T1 variant

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
1  if !HaveDSPExt() then UNDEFINED;
2  d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
3  if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the second general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

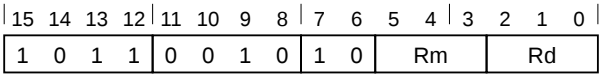
```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      rotated = ROR(R[m], rotation);
4      bits(32) result;
5      result<15:0>  = ZeroExtend(rotated<7:0>,   16);
6      result<31:16> = ZeroExtend(rotated<23:16>, 16);
7      R[d] = result;
```

## C2.4.281 UXTH

Unsigned Extend Halfword. Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

### T1

*Armv8-M*




#### T1 variant

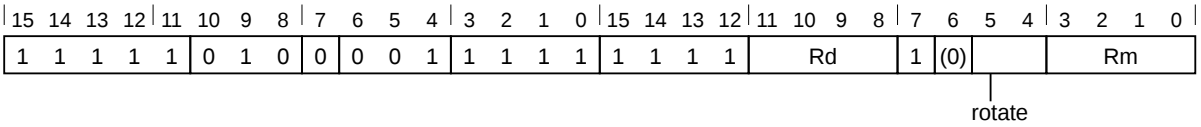
```
UXTH{<c>}{<q>} {<Rd>,} <Rm>
```

### Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

```
UXTH{<c>}.W {<Rd>,} <Rm>
  // <Rd>, <Rm> can be represented in T1
UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rd> | Is the general-purpose destination register, encoded in the "Rd" field. |
| <Rm> | Is the general-purpose source register, encoded in the "Rm" field. |
| <amount> | Is the rotate amount, encoded in the "rotate" field. It can have the following values:<br>0 when rotate = 00<br>8 when rotate = 01<br>16 when rotate = 10<br>24 when rotate = 11<br>ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == 0b00. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

## C2.4.282 VABAV

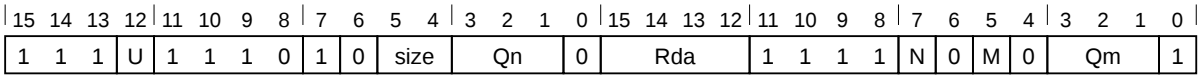
Vector Absolute Difference and Accumulate Across Vector. Subtract the elements of the second source vector register from the corresponding elements of the first source vector and accumulate the absolute values of the results. The initial value of the general-purpose destination register is added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VABAV variant

```
VABAV<v>.<dt> Rda, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' || N == '1' then UNDEFINED;
4  da       = UInt(Rda);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  unsigned = (U == '1');
10 if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
11 if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Rda> | General-purpose source and destination register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
```

```
5
6  op1    = Q[n, curBeat];
7  op2    = Q[m, curBeat];
8  result = UInt(R[da]);
9  for e = 0 to elements-1
10     if elmtMask<e*(esize>>3)> == '1' then
11         result = result + Abs(Int(Elem[op1, e, esize], unsigned) -
12                               Int(Elem[op2, e, esize], unsigned));
13
14 R[da] = result<31:0>;
```

## C2.4.283 VABD (floating-point)

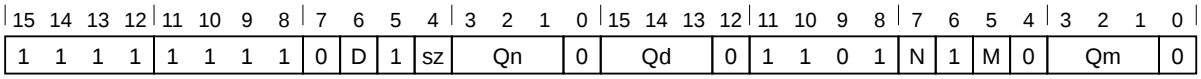
Vector Absolute Difference. Subtract the elements of the second source vector from the corresponding elements of the first source vector and place the absolute values of the results in the elements of the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VABD variant

```
VABD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d        = UInt(D:Qd);
4  m        = UInt(M:Qm);
5  n        = UInt(N:Qn);
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[n, curBeat];
8  op2    = Q[m, curBeat];
9  for e = 0 to elements-1
10     value = FPAbs(FPSub(Elem[op1, e, esize], Elem[op2, e, esize], FALSE));
11     Elem[result, e, esize] = value;
12
13 for e = 0 to 3
14     if elmtMask<e> == '1' then
15         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.284 VABD

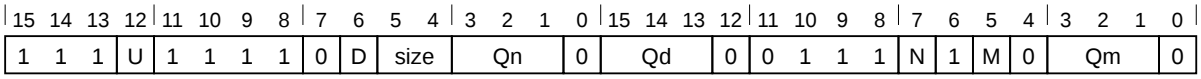
Vector Absolute Difference. Subtract the elements of the second source vector register from the corresponding elements of the first source vector register and place the absolute values of the results in the elements of the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VABD variant

```
VABD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  unsigned = (U == '1');
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
5
```

```
result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
for e = 0 to elements-1
    value = Abs(Int(Elem[op1, e, esize], unsigned) - Int(Elem[op2, e, esize], unsigned));
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.285 VABS (floating-point)

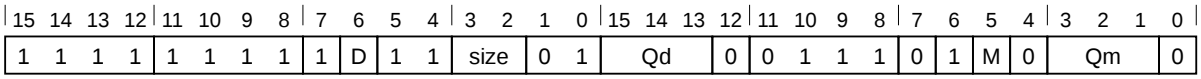
Vector Absolute. Compute the absolute value of each element of a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VABS variant

```
VABS<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' then UNDEFINED;
3  if size IN {'11', '00'} then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F16 Encoded as size = 01<br>F32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = FPAbs(Elem[op1, e, esize]);
10     Elem[result, e, esize] = value;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.286 VABS (vector)

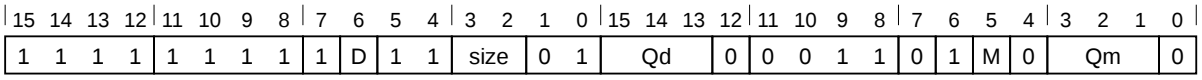
Vector Absolute. Compute the absolute value of each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VABS variant

```
VABS<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

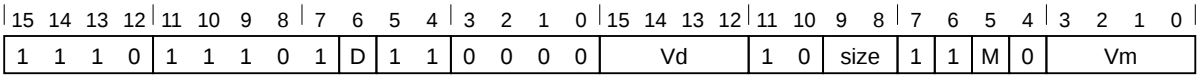
```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = Abs(SInt(Elem[op1, e, esize]));
10     Elem[result, e, esize] = value<esize-1:0>;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.287 VABS

Floating-point Absolute. Floating-point Absolute takes the absolute value of a half-precision or single-precision or double-precision register, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VABS{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4      case size of
5          when '01' S[d] = Zeros(16) : FPAbs(S[m]<15:0>);
6          when '10' S[d] = FPAbs(S[m]);
```

```
7           when '11' D[d] = FPAbs(D[m]);
```

## C2.4.288 VADC

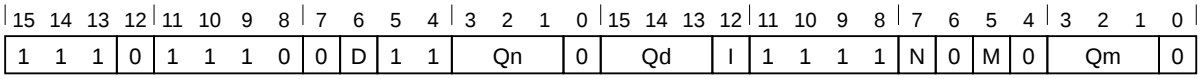
Whole Vector Add With Carry. Add with carry across beats, with carry in from and out to FPSCR.C. Initial value of FPSCR.C can be overridden by using the I variant. FPSCR.C is not updated for beats disabled because of predication. FPSCR.N, .V and .Z are zeroed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VADC variant

```
VADC{I}<v>.I32 Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d         = UInt(D:Qd);
4  m         = UInt(M:Qm);
5  n         = UInt(N:Qn);
6  carryInit = (I == '1');
7  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<I> Specifies where the initial carry in for wide arithmetic comes from. This parameter must be one of the following values:

" Encoded as `I = 0`
Indicates carry input comes from FPSCR.C.

I Encoded as `I = 1`
Indicates carry input is 0.

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Qd> Destination vector register.

<Qn> Source vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  op1 = Q[n, curBeat];
7  op2 = Q[m, curBeat];
8  if carryInit && IsFirstBeat() then
9      (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = ('0', '0', '0', '0');
10 (result, carryOut, -) = AddWithCarry(op1, op2, FPSCR.C);
11 if elmtMask<0> == '1' then
12     (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = ('0', '0', carryOut, '0');
13
```

```
14  for e = 0 to 3
15      if elmtMask<e> == '1' then
16          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.289 VADD (floating-point)

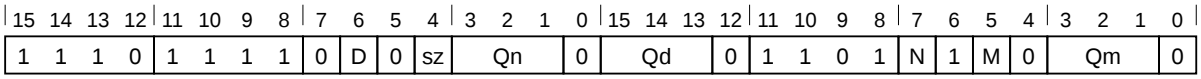
Vector Add. Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VADD variant

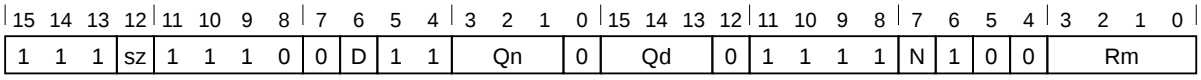
```
VADD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(M:Qm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = FALSE;
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VADD variant

```
VADD<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(Rm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = TRUE;
9  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
10 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = FPAdd(Elem[op1, e, esize], R[m]<esize-1:0>, FALSE);
        Elem[result, e, esize] = value;
else
    for e = 0 to elements-1
        op2   = Q[m, curBeat];
        value = FPAdd(Elem[op1, e, esize], Elem[op2, e, esize], FALSE);
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.290 VADD (vector)

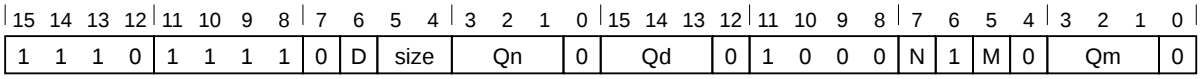
Vector Add. Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is then written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VADD variant

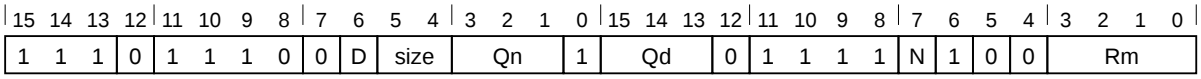
```
VADD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VADD variant

```
VADD<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  withScalar = TRUE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>I8 Encoded as size = 00<br>I16 Encoded as size = 01<br>I32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

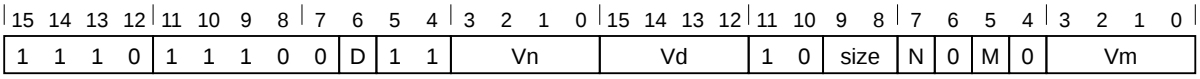
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = Elem[op1, e, esize] + R[m]<esize-1:0>;
        Elem[result, e, esize] = value;
else
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        value = Elem[op1, e, esize] + Elem[op2, e, esize];
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.291 VADD

Floating-point Add. Floating-point Add adds two half-precision or single-precision or double-precision registers, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VADD{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01' S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, TRUE);
        when '10' S[d] = FPAdd(S[n], S[m], TRUE);
        when '11' D[d] = FPAdd(D[n], D[m], TRUE);
```

## C2.4.292 VADDLV

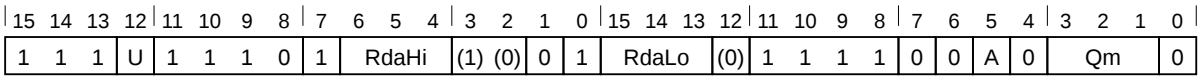
Vector Add Long Across Vector. Add across the elements of a vector accumulating the result into a scalar. The 64 bit result is stored across two registers, the upper-half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VADDLV variant

```
VADDLV{A}<v>.<dt> RdaLo, RdaHi, Qm
```

### Decode for this encoding

```
1  if RdaHi == '111' then SEE "VADDV";
2  CheckDecodeFaults(ExtType_Mve);
3  dah        = UInt(RdaHi:'1');
4  dal        = UInt(RdaLo:'0');
5  m          = UInt(Qm);
6  accumulate = (A == '1');
7  esize      = 32;
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
11 if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <A> | Accumulate with existing register contents.<br>This parameter must be one of the following values:<br>" Encoded as A = 0<br>A Encoded as A = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Unsigned flag: S indicates signed, U indicates unsigned.<br>This parameter must be one of the following values:<br>S32 Encoded as U = 0<br>U32 Encoded as U = 1 |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
```

```
(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then Int(R[dah]:R[dal], unsigned) else 0;
op     = Q[m, curBeat];

for e = 0 to elements-1
    if elmtMask<e*(esize>>3)> == '1' then
        result = result + Int(Elem[op, e, esize], unsigned);

R[dah] = result<63:32>;
R[dal] = result<31:0>;
```

## C2.4.293 VADDV

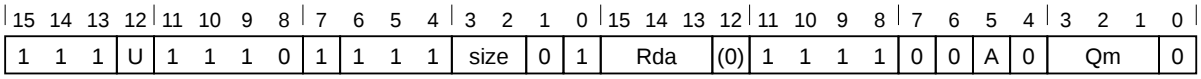
Vector Add Across Vector. Add across the elements of a vector accumulating the result into a scalar. The initial value of the general-purpose destination register can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VADDV variant

```
VADDV{A}<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1 if size == '11' then SEE "Related encodings";
2 CheckDecodeFaults(ExtType_Mve);
3 da          = UInt(Rda:'0');
4 m           = UInt(Qm);
5 accumulate = (A == '1');
6 esize       = 8 << UInt(size);
7 elements    = 32 DIV esize;
8 unsigned    = (U == '1');
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <A> | Accumulate with existing register contents.<br>This parameter must be one of the following values:<br>" Encoded as A = 0<br>A Encoded as A = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Rda> | General-purpose source and destination register. This must be an even numbered register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
```

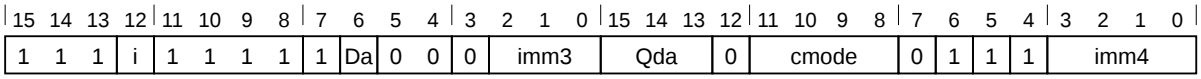
```
5
6  result = if accumulate || !IsFirstBeat() then Int(R[da], unsigned) else 0;
7  op     = Q[m, curBeat];
8
9  for e = 0 to elements-1
10     if elmtMask<e*(esize>>3)> == '1' then
11         result = result + Int(Elem[op, e, esize], unsigned);
12
13 R[da] = result<31:0>;
```

## C2.4.294 VAND (immediate)

Vector Bitwise AND. This is a pseudo-instruction, equivalent to a VBIC (immediate) instruction with the immediate value bitwise inverted.

This is an alias of VBIC (immediate).

### T1




**VAND variant**

```
VAND<v>.<dt> Qda, #<imm>
```

**is equivalent to**

```
VBIC<v>.<dt> Qda, #~<imm>
```

**and is never the preferred disassembly**

## C2.4.295 VAND

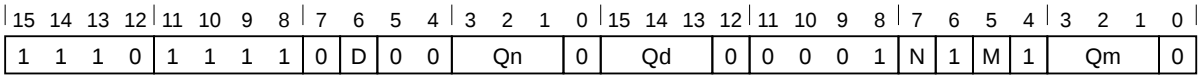
Vector Bitwise And. Compute a bitwise AND of a vector register with another vector register. The result is written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VAND variant

```
VAND<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  n = UInt(N:Qn);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Q[n, curBeat] AND Q[m, curBeat];
7
8  for e = 0 to 3
9      if elmtMask<e> == '1' then
10         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.296 VBIC (immediate)

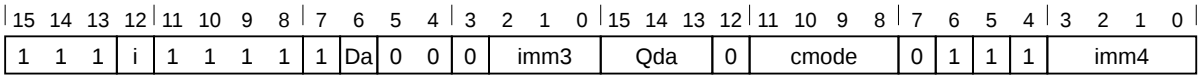
Vector Bitwise Clear. Compute a bitwise AND of a vector register and the complement of an immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VBIC variant

```
VBIC<v>.<dt> Qda, #<imm>
```

### Decode for this encoding

```
1  if cmode == '1110'                          then SEE "VMOV (immediate) (vector)";
2  if cmode IN {'0xx0', 'x0x0', 'xx00', '1101'} then SEE "VMVN (immediate)";
3  CheckDecodeFaults(ExtType_Mve);
4  if Da    == '1'    then UNDEFINED;
5  if cmode == '11x1' then UNDEFINED;
6  da    = UInt(Da:Qda);
7  imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector, for use with the AdvSIMDExpandImm() function.
This parameter must be one of the following values:

- I32 Encoded as:
  - cmode = 0001
  - cmode = 0011
  - cmode = 0101
  - cmode = 0111
- I16 Encoded as:
  - cmode = 1001
  - cmode = 1011

<Qda> Source and destination vector register.

<imm> The immediate value to load in to each element. This must be an immediate that can be encoded for use with the AdvSIMDExpandImm() function.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  opd    = Q[da, curBeat];
7  imm32  = if curBeat<0> == '0' then imm64<31:0> else imm64<63:32>;
8  result = opd AND NOT(imm32);
9
```

```
10  for e = 0 to 3
11      if elmtMask<e> == '1' then
12          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.297 VBIC (register)

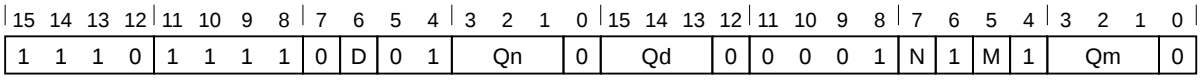
Vector Bitwise Clear. Compute a bitwise AND of a vector register and the complement of a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VBIC variant

```
VBIC<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' || N == '1' then UNDEFINED;
d = UInt(D:Qd);
m = UInt(M:Qm);
n = UInt(N:Qn);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

opm    = Q[m, curBeat];
opn    = Q[n, curBeat];
result = opn AND NOT(opm);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.298 VBRSR

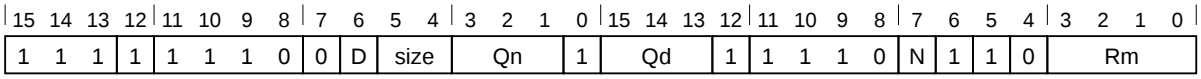
Vector Bit Reverse and Shift Right. Reverse the specified number of LSB bits in each element of a vector register and set the other bits to zero. The number of bits to reverse is read in from the bottom byte of Rm and clamped to the range [0, <dt>].

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VBRSR variant

```
VBRSR<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || N == '1' then UNDEFINED;
d        = UInt(D:Qd);
m        = UInt(Rm);
n        = UInt(N:Qn);
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as size = 00<br>16 Encoded as size = 01<br>32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Rm> | General-purpose register containing the number of LSB bits to reverse in its bottom 8 bits. |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
revBit = UInt(R[m]<7:0>);
for e = 0 to elements-1
    Elem[result, e, esize] = BitReverseShiftRight(Elem[op1, e, esize], revBit);

for e = 0 to 3
```

```
13      if elmtMask<e> == '1' then
14          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.299 VCADD (floating-point)

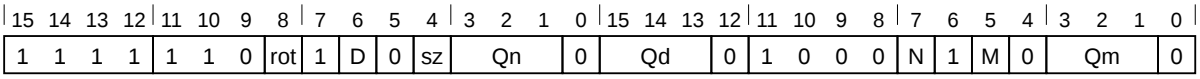
Vector Complex Add with Rotate. This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, while a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components, respectively, of a complex number. The results are written into the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCADD variant

```
VCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d        = UInt(D:Qd);
4  m        = UInt(M:Qm);
5  n        = UInt(N:Qn);
6  esize    = if sz == '0' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()               then CONSTRAINED_UNPREDICTABLE;
9  if D:Qd == M:Qm && sz == '1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F16 Encoded as sz = 0<br>F32 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <rotate> | The rotation amount.<br>This parameter must be one of the following values:<br>#90 Encoded as rot = 0<br>#270 Encoded as rot = 1 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
```

```
if esize == 32 then
    case rot:curBeat<0> of
        when '00' result = FPSub(Q[n, curBeat], Q[m, curBeat+1], FALSE);
        when '01' result = FPAdd(Q[n, curBeat], Q[m, curBeat-1], FALSE);
        when '10' result = FPAdd(Q[n, curBeat], Q[m, curBeat+1], FALSE);
        when '11' result = FPSub(Q[n, curBeat], Q[m, curBeat-1], FALSE);
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        case rot:e<0> of
            when '00' value = FPSub(Elem[op1, e, esize], Elem[op2, e+1, esize], FALSE);
            when '01' value = FPAdd(Elem[op1, e, esize], Elem[op2, e-1, esize], FALSE);
            when '10' value = FPAdd(Elem[op1, e, esize], Elem[op2, e+1, esize], FALSE);
            when '11' value = FPSub(Elem[op1, e, esize], Elem[op2, e-1, esize], FALSE);
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.300 VCADD

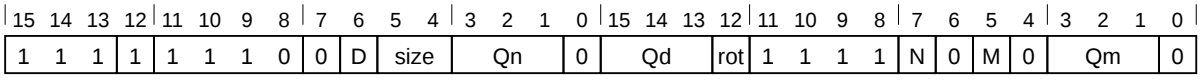
Vector Complex Add with Rotate. This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, while a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components, respectively, of a complex number. The result is then written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VCADD variant

```
VCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  if InITBlock()                  then CONSTRAINED_UNPREDICTABLE;
10 if D:Qd == M:Qm && size == '10' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>I8 Encoded as size = 00<br>I16 Encoded as size = 01<br>I32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <rotate> | The rotation amount.<br>This parameter must be one of the following values:<br>#90 Encoded as rot = 0<br>#270 Encoded as rot = 1 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
```

```
5
6  result = Zeros(32);
7  // 32 bit operations are handled differently as they perform cross beat
8  // register accesses
9  if esize == 32 then
10     case rot:curBeat<0> of
11         when '00' result = (Q[n, curBeat] - Q[m, curBeat+1])<31:0>;
12         when '01' result = (Q[n, curBeat] + Q[m, curBeat-1])<31:0>;
13         when '10' result = (Q[n, curBeat] + Q[m, curBeat+1])<31:0>;
14         when '11' result = (Q[n, curBeat] - Q[m, curBeat-1])<31:0>;
15 else
16     op1 = Q[n, curBeat];
17     op2 = Q[m, curBeat];
18     for e = 0 to elements-1
19         case rot:e<0> of
20             when '00' value = Elem[op1, e, esize] - Elem[op2, e+1, esize];
21             when '01' value = Elem[op1, e, esize] + Elem[op2, e-1, esize];
22             when '10' value = Elem[op1, e, esize] + Elem[op2, e+1, esize];
23             when '11' value = Elem[op1, e, esize] - Elem[op2, e-1, esize];
24         Elem[result, e, esize] = value<esize-1:0>;
25
26 for e = 0 to 3
27     if elmtMask<e> == '1' then
28         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.301 VCLS

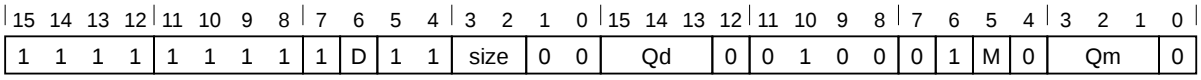
Vector Count Leading Sign-bits. Count the leading sign bits of each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VCLS variant

```
VCLS<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = CountLeadingSignBits(Elem[op1, e, esize]);
10     Elem[result, e, esize] = value<esize-1:0>;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.302 VCLZ

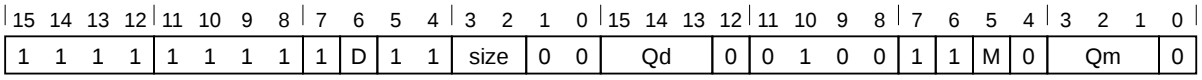
Vector Count Leading Zeros. Count the leading zeros of each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VCLZ variant

```
VCLZ<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D    == '1' || M == '1' then UNDEFINED;
if size == '11'            then UNDEFINED;
d        = UInt(D:Qd);
m        = UInt(M:Qm);
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>I8 Encoded as size = 00<br>I16 Encoded as size = 01<br>I32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    value = CountLeadingZeroBits(Elem[op1, e, esize]);
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.303 VCMLA (floating-point)

Vector Complex Multiply Accumulate. This instruction operates on complex numbers that are represented in registers as pairs of elements. Each element holds a floating-point value. The odd element holds the imaginary part of the number, and the even element holds the real part of the number. The instruction performs the computation on the corresponding complex number element pairs from the two source registers and the destination register. Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees. If the transformation was a rotation by 0 or 180 degrees, the two elements of the transformed complex number are multiplied by the real element of the first source register. If the transformation was a rotation by 90 or 270 degrees, the two elements are multiplied by the imaginary element of the complex number from the first source register. The result of the multiplication is added on to the existing value in the destination vector register. The multiplication and addition operations are fused and the result is not rounded.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | rot | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | | | | | | Da | | 1 | sz | Qn | | | 0 | Qda | | | 0 | 1 | 0 | 0 | 0 | N | | M | 0 | Qm | | | 0 |

#### T1: VCMLA variant

```
VCMLA<v>.<dt> Qda, Qn, Qm, #<rotate>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if Da == '1' || M == '1' || N == '1' then UNDEFINED;
3 da       = UInt(Da:Qda);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 esize    = if sz == '0' then 16 else 32;
7 elements = 32 DIV esize;
8 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
9 if sz == '1' && (Da:Qda == M:Qm || Da:Qda == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F16 Encoded as sz = 0<br>F32 Encoded as sz = 1 |
| <Qda> | Source and destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <rotate> | The rotation amount.<br>This parameter must be one of the following values:<br>#0 Encoded as rot = 00<br>#90 Encoded as rot = 01<br>#180 Encoded as rot = 10<br>#270 Encoded as rot = 11 |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
dest   = Q[da, curBeat];
if esize == 32 then
    if (curBeat<0> == '0') then
        case rot of
            when '00'
                element1 =       Q[m, curBeat];
                element2 =       Q[n, curBeat];
            when '01'
                element1 = FPNeg(Q[m, curBeat+1]);
                element2 =       Q[n, curBeat+1];
            when '10'
                element1 = FPNeg(Q[m, curBeat]);
                element2 =       Q[n, curBeat];
            when '11'
                element1 =       Q[m, curBeat+1];
                element2 =       Q[n, curBeat+1];
    else
        case rot of
            when '00'
                element1 =       Q[m, curBeat];
                element2 =       Q[n, curBeat-1];
            when '01'
                element1 =       Q[m, curBeat-1];
                element2 =       Q[n, curBeat];
            when '10'
                element1 = FPNeg(Q[m, curBeat]);
                element2 =       Q[n, curBeat-1];
            when '11'
                element1 = FPNeg(Q[m, curBeat-1]);
                element2 =       Q[n, curBeat];
    result = FPMulAdd(dest, element2, element1, FALSE);
else
    op1 = Q[m, curBeat];
    op2 = Q[n, curBeat];
    case rot of
        when '00'
            elem1 =       Elem[op1, 0, esize];
            elem2 =       Elem[op2, 0, esize];
            elem3 =       Elem[op1, 1, esize];
            elem4 =       Elem[op2, 0, esize];
        when '01'
            elem1 = FPNeg(Elem[op1, 1, esize]);
            elem2 =       Elem[op2, 1, esize];
            elem3 =       Elem[op1, 0, esize];
            elem4 =       Elem[op2, 1, esize];
        when '10'
            elem1 = FPNeg(Elem[op1, 0, esize]);
            elem2 =       Elem[op2, 0, esize];
            elem3 = FPNeg(Elem[op1, 1, esize]);
            elem4 =       Elem[op2, 0, esize];
        when '11'
            elem1 =       Elem[op1, 1, esize];
            elem2 =       Elem[op2, 1, esize];
            elem3 = FPNeg(Elem[op1, 0, esize]);
            elem4 =       Elem[op2, 1, esize];
    Elem[result, 0, esize] = FPMulAdd(Elem[dest, 0, esize], elem2, elem1, FALSE);
    Elem[result, 1, esize] = FPMulAdd(Elem[dest, 1, esize], elem4, elem3, FALSE);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.304 VCMP (floating-point)

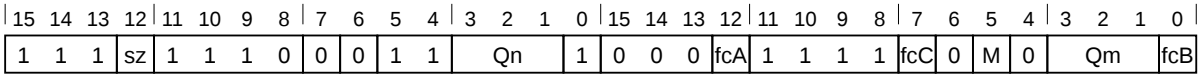
Vector Compare. Perform a lane-wise comparison between each element in the first source vector register and either the respective elements in the second source vector register or the value of a general-purpose register. The resulting boolean conditions are placed in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCMP variant

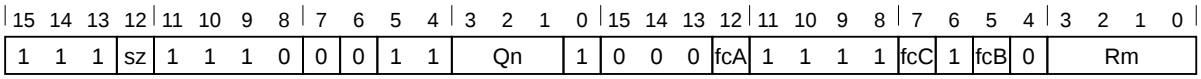
```
VCMP<v>.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if fcA == '0' && fcB == '1' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_MveFp);
3  if M == '1' then UNDEFINED;
4  m          = UInt(M:Qm);
5  n          = UInt(Qn);
6  fcond      = fcA:fcB:fcC;
7  withScalar = FALSE;
8  esize      = 8 << UInt(if sz == '1' then '01' else '10');
9  elements   = 32 DIV esize;
10 ebytes     = esize DIV 8;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VCMP variant

```
VCMP<v>.<dt> <fc>, Qn, Rm
```

### Decode for this encoding

```
1  if Rm == '1101' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_MveFp);
3  m          = UInt(Rm);
4  n          = UInt(Qn);
5  fcond      = fcA:fcB:fcC;
6  withScalar = TRUE;
7  esize      = 8 << UInt(if sz == '1' then '01' else '10');
8  elements   = 32 DIV esize;
9  ebytes     = esize DIV 8;
10 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
11 if fcA == '0' && fcB == '1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <fc> | The comparison condition to use.<br>This parameter must be one of the following values:<br>EQ Encoded as fcA = 0, fcB = 0, fcC = 0<br>NE Encoded as fcA = 0, fcB = 0, fcC = 1<br>GE Encoded as fcA = 1, fcB = 0, fcC = 0<br>LT Encoded as fcA = 1, fcB = 0, fcC = 1<br>GT Encoded as fcA = 1, fcB = 1, fcC = 0<br>LE Encoded as fcA = 1, fcB = 1, fcC = 1 |
| <Qn> | First source vector register |
| <Qm> | Source vector register. |
| <Rm> | Source general-purpose register (ZR is permitted, PC is not). |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1      = Q[n, curBeat];
beatPred = Zeros(4);
if withScalar then
    op2 = RZ[m]<esize-1:0>;
else
    opm = Q[m, curBeat];
for e = 0 to elements-1
    if !withScalar then
        op2 = Elem[opm, e, esize];
    (flN, flZ, flC, flV) = FPCompare(Elem[op1, e, esize], op2, TRUE, TRUE);
    pred = ConditionHolds(fcond, flN, flZ, flC, flV);
    Elem[beatPred, e, ebytes] = Replicate(if pred then '1' else '0');

Elem[VPR.P0, curBeat, 4] = beatPred AND elmtMask;
```

## C2.4.305 VCMP (vector)

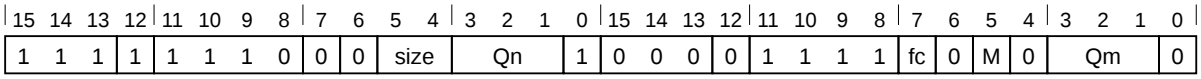
Vector Compare. Perform a lane-wise comparison between each element in the first source vector register and either the respective elements in the second source vector register or the value of a general-purpose register. The resulting boolean conditions are placed in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VCMP variant

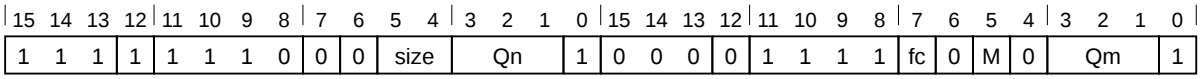
```
VCMP<v>.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  m          = UInt(M:Qm);
5  n          = UInt(Qn);
6  fcond      = '00':fc;
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 ebytes     = esize DIV 8;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VCMP variant

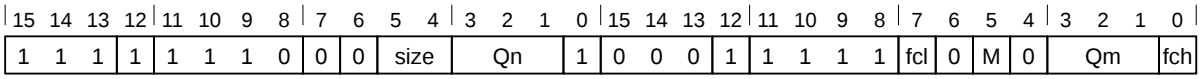
```
VCMP<v>.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  m          = UInt(M:Qm);
5  n          = UInt(Qn);
6  fcond      = '01':fc;
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 ebytes     = esize DIV 8;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T3

*Armv8.1-M MVE*




#### T3: VCMP variant

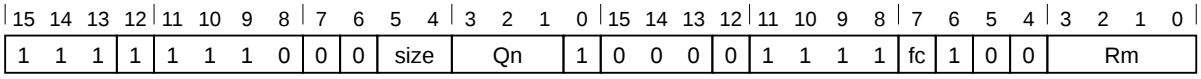
```
VCMP<v>.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  m          = UInt(M:Qm);
5  n          = UInt(Qn);
6  fcond      = '1':fch:fcl;
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 ebytes     = esize DIV 8;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T4

*Armv8.1-M MVE*




#### T4: VCMP variant

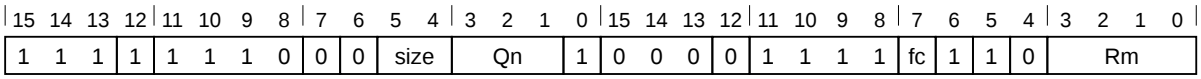
```
VCMP<v>.<dt> <fc>, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  m          = UInt(Rm);
4  n          = UInt(Qn);
5  fcond      = '00':fc;
6  withScalar = TRUE;
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  ebytes     = esize DIV 8;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

### T5

*Armv8.1-M MVE*




#### T5: VCMP variant

```
VCMP<v>.<dt> <fc>, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  m          = UInt(Rm);
4  n          = UInt(Qn);
5  fcond      = '01':fc;
6  withScalar = TRUE;
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  ebytes     = esize DIV 8;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | size | | Qn | | | | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | fcl | 1 | fch | 0 | Rm | | | |

### T6: VCMP variant

```
VCMP<v>.<dt> <fc>, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  m          = UInt(Rm);
4  n          = UInt(Qn);
5  fcond      = '1':fch:fcl;
6  withScalar = TRUE;
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  ebytes     = esize DIV 8;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `I8` Encoded as `size = 00`
- `I16` Encoded as `size = 01`
- `I32` Encoded as `size = 10`

`<fc>` The comparison condition to use.
This parameter must be one of the following values:

- `EQ` Encoded as `fc = 0`
- `NE` Encoded as `fc = 1`

### Assembler symbols for T2 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `U8` Encoded as `size = 00`
- `U16` Encoded as `size = 01`

U32 Encoded as size = 10

<fc> The comparison condition to use.
This parameter must be one of the following values:
CS Encoded as fc = 0
HI Encoded as fc = 1

### Assembler symbols for T3 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
S8 Encoded as size = 00
S16 Encoded as size = 01
S32 Encoded as size = 10

<fc> The comparison condition to use.
This parameter must be one of the following values:
GE Encoded as fch = 0, fcl = 0
LT Encoded as fch = 0, fcl = 1
GT Encoded as fch = 1, fcl = 0
LE Encoded as fch = 1, fcl = 1

### Assembler symbols for T4 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
I8 Encoded as size = 00
I16 Encoded as size = 01
I32 Encoded as size = 10

<fc> The comparison condition to use.
This parameter must be one of the following values:
EQ Encoded as fc = 0
NE Encoded as fc = 1

### Assembler symbols for T5 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
U8 Encoded as size = 00
U16 Encoded as size = 01
U32 Encoded as size = 10

<fc> The comparison condition to use.
This parameter must be one of the following values:
CS Encoded as fc = 0
HI Encoded as fc = 1

### Assembler symbols for T6 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
S8 Encoded as size = 00
S16 Encoded as size = 01
S32 Encoded as size = 10

<fc> The comparison condition to use.

This parameter must be one of the following values:

| | | | |
|---|---|---|---|
| GE | Encoded as | fch = 0, | fcl = 0 |
| LT | Encoded as | fch = 0, | fcl = 1 |
| GT | Encoded as | fch = 1, | fcl = 0 |
| LE | Encoded as | fch = 1, | fcl = 1 |

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qn> | First source vector register |
| <Qm> | Second source vector register |
| <Rm> | Source general-purpose register (ZR is permitted, PC is not). |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1      = Q[n, curBeat];
beatPred = Zeros(4);
if withScalar then
    op2 = RZ[m]<esize-1:0>;
else
    opm = Q[m, curBeat];
for e = 0 to elements-1
    if !withScalar then
        op2 = Elem[opm, e, esize];
    (result, flC, flV) = AddWithCarry(Elem[op1, e, esize], NOT(op2), '1');
    flN                = result<esize-1>;
    flZ                = IsZeroBit(result);
    pred = ConditionHolds(fcond, flN, flZ, flC, flV);
    Elem[beatPred, e, ebytes] = Replicate(if pred then '1' else '0');

Elem[VPR.P0, curBeat, 4] = beatPred AND elmtMask;
```

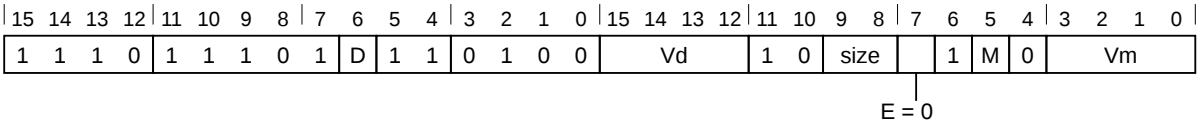
## C2.4.306 VCMP

Floating-point Compare. Floating-point Compare compares two registers, or one register and zero. It writes the result to FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent VMRS instruction.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCMP{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

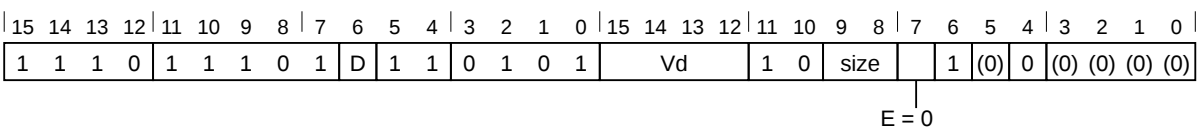
```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (size == '11');
2 CheckFPDecodeFaults(size);
3 if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4 if size == '01' && InITBlock() then UNPREDICTABLE;
5 quiet_nan_exc = (E == '1');  with_zero = FALSE;
6 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T2

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

## Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = integer UNKNOWN;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if with_zero then FPZero('0',16) else S[m]<15:0>;
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d]<15:0>, op16, quiet_nan_exc,
                 TRUE);
        when '10'
            op32 = if with_zero then FPZero('0',32) else S[m];
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op32, quiet_nan_exc, TRUE)
                ;
        when '11'
            op64 = if with_zero then FPZero('0',64) else D[m];
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op64, quiet_nan_exc, TRUE)
                ;
```

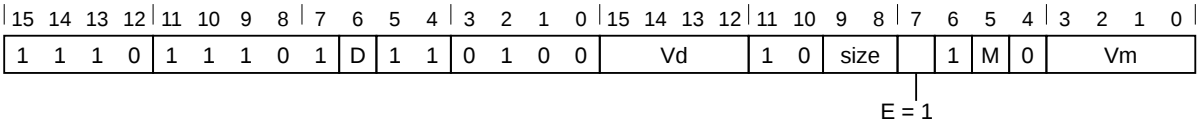
## C2.4.307 VCMPE

Floating-point Compare, raising Invalid Operation on NaN. Floating-point Compare, raising Invalid Operation on NaN compares two registers, or one register and zero. It writes the result to FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent VMRS instruction.

It raises an Invalid Operation exception if either operand is any type of NaN.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

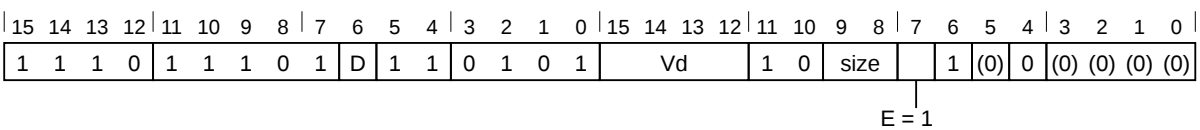
```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (size == '11');
2 CheckFPDecodeFaults(size);
3 if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4 if size == '01' && InITBlock() then UNPREDICTABLE;
5 quiet_nan_exc = (E == '1');  with_zero = FALSE;
6 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### T2

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCMPE{<c>}{<q>}.F16 <Sd>, #0.0
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

## Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1');  with_zero = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = integer UNKNOWN;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if with_zero then FPZero('0',16) else S[m]<15:0>;
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d]<15:0>, op16, quiet_nan_exc,
                 TRUE);
        when '10'
            op32 = if with_zero then FPZero('0',32) else S[m];
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(S[d], op32, quiet_nan_exc, TRUE)
                ;
        when '11'
            op64 = if with_zero then FPZero('0',64) else D[m];
            (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = FPCompare(D[d], op64, quiet_nan_exc, TRUE)
                ;
```

## C2.4.308 VCMUL (floating-point)

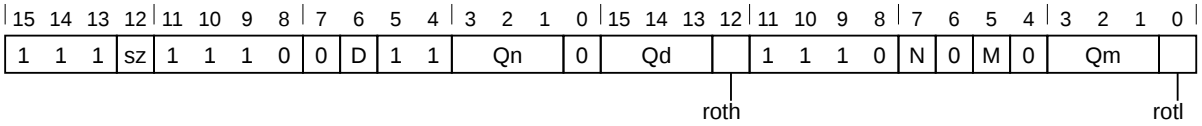
Vector Complex Multiply. This instruction operates on complex numbers that are represented in registers as pairs of elements. Each element holds a floating-point value. The odd element holds the imaginary part of the number, and the even element holds the real part of the number. The instruction performs the computation on the corresponding complex number element pairs from the two source registers and the destination register. Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees. If the transformation was a rotation by 0 or 180 degrees, the two elements of the transformed complex number are multiplied by the real element of the first source register. If the transformation was a rotation by 90 or 270 degrees, the two elements are multiplied by the imaginary element of the complex number from the first source register. The results are written into the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCMUL variant

```
VCMUL<v>.<dt> Qd, Qn, Qm, #<rotate>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if D == '1' || M == '1' || N == '1' then UNDEFINED;
3 d        = UInt(D:Qd);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 esize    = if sz == '0' then 16 else 32;
7 elements = 32 DIV esize;
8 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
9 if sz == '1' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F16 Encoded as sz = 0<br>F32 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <rotate> | The rotation amount.<br>This parameter must be one of the following values:<br>#0 Encoded as roth = 0, rotl = 0<br>#90 Encoded as roth = 0, rotl = 1<br>#180 Encoded as roth = 1, rotl = 0<br>#270 Encoded as roth = 1, rotl = 1 |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
if esize == 32 then
    if (curBeat<0> == '0') then
        case roth:rotl of
            when '00'
                element1 =       Q[m, curBeat];
                element2 =       Q[n, curBeat];
            when '01'
                element1 = FPNeg(Q[m, curBeat+1]);
                element2 =       Q[n, curBeat+1];
            when '10'
                element1 = FPNeg(Q[m, curBeat]);
                element2 =       Q[n, curBeat];
            when '11'
                element1 =       Q[m, curBeat+1];
                element2 =       Q[n, curBeat+1];
    else
        case roth:rotl of
            when '00'
                element1 =       Q[m, curBeat];
                element2 =       Q[n, curBeat-1];
            when '01'
                element1 =       Q[m, curBeat-1];
                element2 =       Q[n, curBeat];
            when '10'
                element1 = FPNeg(Q[m, curBeat]);
                element2 =       Q[n, curBeat-1];
            when '11'
                element1 = FPNeg(Q[m, curBeat-1]);
                element2 =       Q[n, curBeat];
    result = FPMul(element2, element1, FALSE);
else
    op1 = Q[m, curBeat];
    op2 = Q[n, curBeat];
    case roth:rotl of
        when '00'
            elem1 =       Elem[op1, 0, esize];
            elem2 =       Elem[op2, 0, esize];
            elem3 =       Elem[op1, 1, esize];
            elem4 =       Elem[op2, 0, esize];
        when '01'
            elem1 = FPNeg(Elem[op1, 1, esize]);
            elem2 =       Elem[op2, 1, esize];
            elem3 =       Elem[op1, 0, esize];
            elem4 =       Elem[op2, 1, esize];
        when '10'
            elem1 = FPNeg(Elem[op1, 0, esize]);
            elem2 =       Elem[op2, 0, esize];
            elem3 = FPNeg(Elem[op1, 1, esize]);
            elem4 =       Elem[op2, 0, esize];
        when '11'
            elem1 =       Elem[op1, 1, esize];
            elem2 =       Elem[op2, 1, esize];
            elem3 = FPNeg(Elem[op1, 0, esize]);
            elem4 =       Elem[op2, 1, esize];
    Elem[result, 0, esize] = FPMul(elem2, elem1, FALSE);
    Elem[result, 1, esize] = FPMul(elem4, elem3, FALSE);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.309 VCTP

Create Vector Tail Predicate. Creates a predicate pattern in VPR.P0 such that any element numbered the value of Rn or greater is predicated. Any element numbered lower than the value of Rn is not predicated. If placed within a VPT block and a lane is predicated, the corresponding VPR.P0 pattern will also be predicated. The generated VPR.P0 pattern can be used by an ensuing predication instruction to apply tail predication on a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 0 | 0 | 0 | 0 | size | | Rn | | | | 1 | 1 | 1 | 0 | 1 | (0) | (0) | (0) | (0) | (0) | (0) | (0) | (0) | (0) | (0) | 1 |

#### T1: VCTP variant

```
VCTP<v>.<dt> Rn
```

### Decode for this encoding

```
1  if Rn == '1111' then SEE "Related encodings";
2  if !HaveMve()   then UNDEFINED;
3  HandleException(CheckCPEnabled(10));
4  n        = UInt(Rn);
5  predSize = UInt(size);
6  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
7  if Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> The size of the elements in the vector to process.
This parameter must be one of the following values:

| | | |
|---|---|---|
| 8 | Encoded as | size = 00 |
| 16 | Encoded as | size = 01 |
| 32 | Encoded as | size = 10 |
| 64 | Encoded as | size = 11 |

<Rn> The register containing the number of elements that need to be processed.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  loopCount = R[n];
7  if UInt(loopCount) <= (1 << (4 - predSize)) then
8      fullMask = ZeroExtend(Ones(UInt(loopCount<4-predSize:0> : Zeros(predSize))), 16);
9  else
10     fullMask = Ones(16);
11
12 Elem[VPR.P0, curBeat, 4] = elmtMask AND Elem[fullMask, curBeat, 4];
```

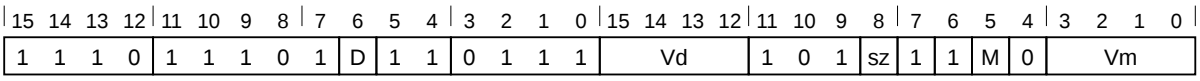
## C2.4.310 VCVT (between double-precision and single-precision)

Convert between double-precision and single-precision. This instruction does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

### T1

*Armv8-M Floating-point Extension only*




**Encoding**

**Applies when `sz == 0`.**

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

**Encoding**

**Applies when `sz == 1`.**

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_DpFp);
2  double_to_single = (sz == '1');
3  if double_to_single then
4      if VFPSmallRegisterBank() && (M == '1') then UNDEFINED;
5      d = UInt(Vd:D);
6      m = UInt(M:Vm);
7  else
8      if VFPSmallRegisterBank() && (D == '1') then UNDEFINED;
9      d = UInt(D:Vd);
10     m = UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();  ExecuteFPCheck();
3      if double_to_single then
4          S[d] = FPDoubleToSingle(D[m], TRUE);
5      else
6          D[d] = FPSingleToDouble(S[m], TRUE);
```

## C2.4.311 VCVT (between floating-point and fixed-point) (vector)

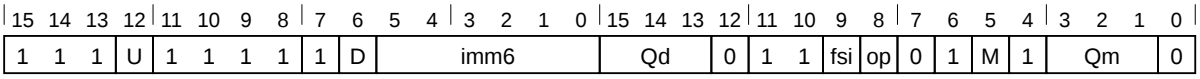
Vector Convert between floating-point and fixed-point. Convert between floating-point and fixed-point values in elements of a vector register. The number of fractional bits in the fixed-point value is specified by an immediate. Fixed-point values can be specified as signed or unsigned. The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode. For floating-point to fixed-point operation, if the source value is outside the range of the target fixed-point type, the result is saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCVT variant

```
VCVT<v>.<dt> Qd, Qm, #<fbits>
```

### Decode for this encoding

```
1  if imm6 == '000xxx' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_MveFp);
3  if D    == '1' || M == '1'                              then UNDEFINED;
4  if imm6 == '0xxxxx' || (fsi == '0' && imm6 == '10xxxx') then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(M:Qm);
7  esize    = 16 << UInt(fsi);
8  elements = 32 DIV esize;
9  toFixed  = (op == '1');
10 unsigned = (U  == '1');
11 fracBits = 64 - UInt(imm6);
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter must be one of the following values:

`F16.S16` Encoded as `fsi = 0, op = 0, U = 0`
Convert signed 16 bit integer to half-precision floating-point

`F16.U16` Encoded as `fsi = 0, op = 0, U = 1`
Convert unsigned 16 bit integer to half-precision floating-point

`S16.F16` Encoded as `fsi = 0, op = 1, U = 0`
Convert half-precision floating-point to signed 16 bit integer

`U16.F16` Encoded as `fsi = 0, op = 1, U = 1`
Convert half-precision floating-point to unsigned 16 bit integer

`F32.S32` Encoded as `fsi = 1, op = 0, U = 0`
Convert signed 32 bit integer to single-precision floating-point

`F32.U32` Encoded as `fsi = 1, op = 0, U = 1`
Convert unsigned 32 bit integer to single-precision floating-point

`S32.F32` Encoded as `fsi = 1, op = 1, U = 0`
Convert single-precision floating-point to signed 32 bit integer

`U32.F32` Encoded as `fsi = 1, op = 1, U = 1`

Convert single-precision floating-point to unsigned 32 bit integer

| | |
|---|---|
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |
| <fbits> | The number of fraction bits in the fixed-point number. For 16-bit fixed-point, this number must be in the range 1-16. For 32-bit fixed-point, this number must be in the range 1-32. The value of (64 - <fbits>) is encoded in imm6. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    if toFixed then
        // Round to zero
        value = FPToFixed(Elem[op1, e, esize], esize, fracBits, unsigned, TRUE, FALSE);
    else
        // Round nearest
        value = FixedToFP(Elem[op1, e, esize], esize, fracBits, unsigned, TRUE, FALSE);
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.312 VCVT (between floating-point and fixed-point)

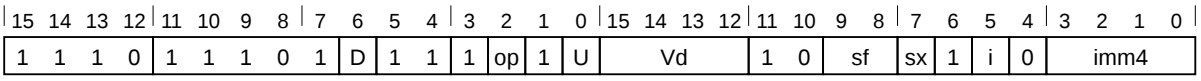
Floating-point Convert (between floating-point and fixed-point). Floating-point Convert (between floating-point and fixed-point) converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point, and places the result in the destination register. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

### T1

*Armv8-M Floating-point Extension only,* sf == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `op == 0 && sf == 01`.**

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `op == 1 && sf == 01`.**

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sf == 10`.**

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sf == 10`.**

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sf == 11`.**

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sf == 11`.**

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

## Decode for this encoding

```
dp_operation = (sf == '11');
CheckFPDecodeFaults(sf);
if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
if sf == '01' && InITBlock() then UNPREDICTABLE;
to_fixed = (op == '1');  unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
if to_fixed then
    round_zero = TRUE;
else
    round_nearest = TRUE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `frac_bits < 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:<br>S16 when U = 0, sx = 0<br>S32 when U = 0, sx = 1<br>U16 when U = 1, sx = 0<br>U32 when U = 1, sx = 1 |
| <Sdm> | Is the 32-bit name of the floating-point destination and source register, encoded in the "Vd:D" field. |
| <Ddm> | Is the 64-bit name of the floating-point destination and source register, encoded in the "D:Vd" field. |
| <fbits> | The number of fraction bits in the fixed-point number:<br>- If <dt> is S16 or U16, <fbits> must be in the range 0-16. (16 - <fbits>) is encoded in [imm4, i]<br>- If <dt> is S32 or U32, <fbits> must be in the range 1-32. (32 - <fbits>) is encoded in [imm4, i]. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_fixed then
        case sf of
            when '01'
```

```
                result = FPToFixed(S[d]<15:0>, size, frac_bits, unsigned, round_zero, TRUE);
                S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
            when '10'
                result = FPToFixed(S[d], size, frac_bits, unsigned, round_zero, TRUE);
                S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
            when '11'
                result = FPToFixed(D[d], size, frac_bits, unsigned, round_zero, TRUE);
                D[d] = if unsigned then ZeroExtend(result, 64) else SignExtend(result, 64);
    else
        case sf of
            when '01'
                fp16 = FixedToFP(S[d]<size-1:0>, 16, frac_bits, unsigned, round_nearest, TRUE
                    );
                S[d] = Zeros(16):fp16;
            when '10'
                S[d] = FixedToFP(S[d]<size-1:0>, 32, frac_bits, unsigned, round_nearest, TRUE
                    );
            when '11'
                D[d] = FixedToFP(D[d]<size-1:0>, 64, frac_bits, unsigned, round_nearest, TRUE
                    );
```

## C2.4.313 VCVT (between floating-point and integer)

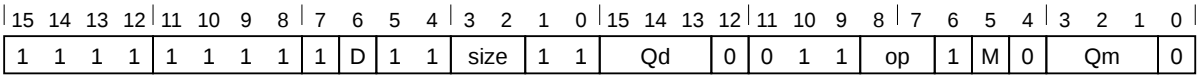
Vector Convert between floating-point and integer. Convert between floating-point and integer values in elements of a vector register. When converting to integer the value is rounded towards zero, when converting to floating-point the value is rounded to nearest. For floating-point to integer operation, if the source value is outside the range of the target integer type, the result is saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCVT variant

```
VCVT<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' then UNDEFINED;
3  if size IN {'11', '00'} then UNDEFINED;
4  d         = UInt(D:Qd);
5  m         = UInt(M:Qm);
6  toInteger = (op<1> == '1');
7  unsigned  = (op<0> == '1');
8  esize     = 8 << UInt(size);
9  elements  = 32 DIV esize;
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter must be one of the following values:

- `F16.S16` Encoded as `op = 00`, `size = 01`
  Convert signed 16 bit integer to half-precision floating-point
- `F32.S32` Encoded as `op = 00`, `size = 10`
  Convert signed 32 bit integer to single-precision floating-point
- `F16.U16` Encoded as `op = 01`, `size = 01`
  Convert unsigned 16 bit integer to half-precision floating-point
- `F32.U32` Encoded as `op = 01`, `size = 10`
  Convert unsigned 32 bit integer to single-precision floating-point
- `S16.F16` Encoded as `op = 10`, `size = 01`
  Convert half-precision floating-point to signed 16 bit integer
- `S32.F32` Encoded as `op = 10`, `size = 10`
  Convert single-precision floating-point to signed 32 bit integer
- `U16.F16` Encoded as `op = 11`, `size = 01`
  Convert half-precision floating-point to unsigned 16 bit integer
- `U32.F32` Encoded as `op = 11`, `size = 10`
  Convert single-precision floating-point to unsigned 32 bit integer

<Qd> Destination vector register.

<Qm> Source vector register.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    if toInteger then
        // Round to zero
        value = FPToFixed(Elem[op1, e, esize], esize, 0, unsigned, TRUE, FALSE);
    else
        // Round to nearest
        value = FixedToFP(Elem[op1, e, esize], esize, 0, unsigned, TRUE, FALSE);
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.314 VCVT (between single and half-precision floating-point)

Vector Convert between half-precision and single-precision. Convert between half-precision and single-precision floating-point values in elements of a vector register. For half-precision to single-precision operation, the top half (T variant) or bottom half (B variant) of the source vector register is selected. For single-precision to half-precision operation, the top half (T variant) or bottom half (B variant) of the destination vector register is selected and the other half retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | op | 1 | 1 | 1 | 0 | 0 | D | 1 | 1 | 1 | 1 | 1 | 1 | Qd | | | | T | 1 | 1 | 1 | 0 | 0 | 0 | M | 0 | Qm | | | 1 |

#### T1: VCVT variant

```
VCVT<T><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  elements     = 4;
6  esize        = 32;
7  halfToSingle = (op == '1');
8  top          = UInt(T);
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies that the FP16 value read from or written to the top (T) or bottom (B) half of the FP32 vector register element.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter must be one of the following values:
- F16.F32 Encoded as op = 0
  Convert single-precision to half-precision
- F32.F16 Encoded as op = 1
  Convert half-precision to single-precision

<Qd> Destination vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
```

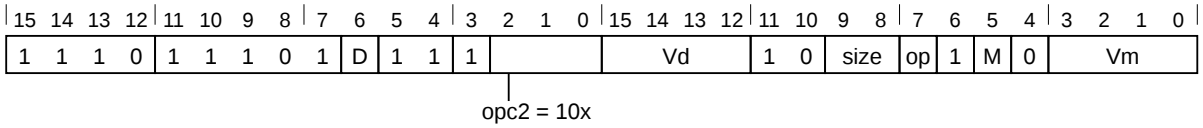
```
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  if halfToSingle then
9      result = FPHalfToSingle(Elem[op1, top, 16], FALSE);
10 else
11     // Write to the selected half of the destination element
12     Elem[result,   top,  16] = FPSingleToHalf(op1, FALSE);
13     // Don't overwrite the other half
14     Elem[elmtMask, 1-top, 2] = '00';
15
16 for e = 0 to 3
17     if elmtMask<e> == '1' then
18         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.315 VCVT (floating-point to integer)

Convert floating-point to integer with Round towards Zero. Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 01 && op == 1`.**

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 01 && op == 0`.**

```
VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 10 && op == 1`.**

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 10 && op == 1`.**

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 11 && op == 1`.**

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 11 && op == 1`.**

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
to_integer = (opc2<2> == '1');
if to_integer then
    if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
    unsigned = (op == '0');  round_nearest = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if size == '01' && InITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        case size of
            when '01'
                S[d] = FPToFixed(S[m]<15:0>, 32, 0, unsigned, round_zero, TRUE);
            when '10'
                S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
            when '11'
                S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
    else
        case size of
            when '01'
                fp16 = FixedToFP(S[m], 16, 0, unsigned, round_nearest, TRUE);
                S[d] = Zeros(16):fp16;
            when '10'
                S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
            when '11'
                D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
```

## C2.4.316 VCVT (from floating-point to integer)

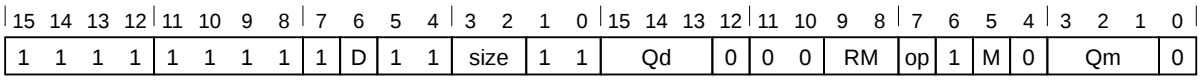
Vector Convert from floating-point to integer. Convert each element in a vector from floating-point to integer using the specified rounding mode and place the results in a second vector. If a source element is outside the range of the target integer type, the result element is saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VCVT variant

```
VCVT<ANPM><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if D == '1' || M == '1' then UNDEFINED;
3 if size IN {'11', '00'} then UNDEFINED;
4 d        = UInt(D:Qd);
5 m        = UInt(M:Qm);
6 unsigned = (op == '1');
7 esize    = 8 << UInt(size);
8 elements = 32 DIV esize;
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<ANPM> The rounding mode.
This parameter must be one of the following values:
- A Encoded as RM = 00
  Round to nearest with ties to away
- N Encoded as RM = 01
  Round to nearest with ties to even
- P Encoded as RM = 10
  Round towards plus infinity
- M Encoded as RM = 11
  Round towards minus infinity

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter must be one of the following values:
- S16.F16 Encoded as op = 0, size = 01
  Convert half-precision floating-point to signed 16 bit integer
- S32.F32 Encoded as op = 0, size = 10
  Convert single-precision floating-point to signed 32 bit integer
- U16.F16 Encoded as op = 1, size = 01
  Convert half-precision floating-point to unsigned 16 bit integer
- U32.F32 Encoded as op = 1, size = 10
  Convert single-precision floating-point to unsigned 32 bit integer

<Qd> Destination vector register.

<Qm> Source vector register.

## Operation for all encodings

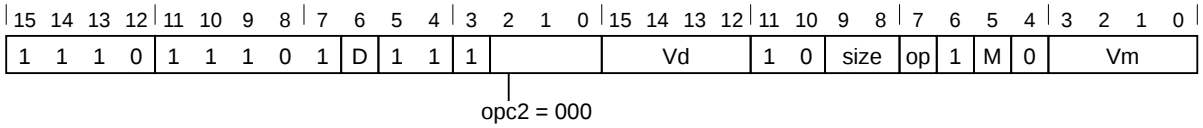
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    Elem[result, e, esize] = FPToFixedDirected(Elem[op1, e, esize], 0, unsigned, RM, FALSE);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.317 VCVT (integer to floating-point)

Convert integer to floating-point. Convert integer to floating-point converts a value in a register from a 32-bit integer to floating-point, using the rounding mode specified by FPSCR, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

### Decode for this encoding

```
1  if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
2  dp_operation = (size == '11');
3  CheckFPDecodeFaults(size);
4  to_integer = (opc2<2> == '1');
5  if to_integer then
6      if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
7      unsigned = (opc2<0> == '0');  round_zero = (op == '1');
8      d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
9  else
10     if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
11     unsigned = (op == '0');  round_nearest = FALSE;  // FALSE selects FPSCR rounding
12     m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
13 if size == '01' && InITBlock() then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the operand, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |

<Sd> Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
<Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
<Sm> Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

## Operation for all encodings

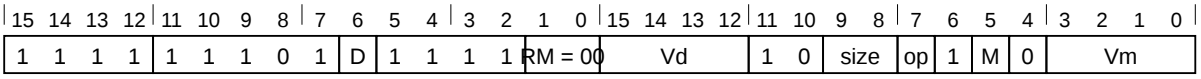
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        case size of
            when '01'
                S[d] = FPToFixed(S[m]<15:0>, 32, 0, unsigned, round_zero, TRUE);
            when '10'
                S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
            when '11'
                S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
    else
        case size of
            when '01'
                fp16 = FixedToFP(S[m], 16, 0, unsigned, round_nearest, TRUE);
                S[d] = Zeros(16):fp16;
            when '10'
                S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
            when '11'
                D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
```

## C2.4.318 VCVTA

Convert floating-point to integer with Round to Nearest with Ties to Away. Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  unsigned = (op == '0');
6  round_mode = RM;
7  d = UInt(Vd:D);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

case size of
    when '01'
        S[d] = FPToFixedDirected(S[m]<15:0>, 0, unsigned, round_mode, TRUE);
    when '10'
        S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);
    when '11'
        S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
```

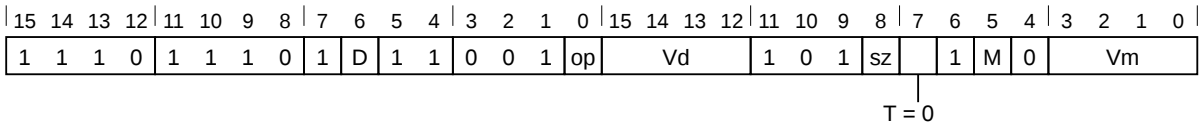
## C2.4.319 VCVTB

Floating-point Convert Bottom. Floating-point Convert Bottom does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the target register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register, without intermediate rounding.
- Converts the value in the double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the target register, without intermediate rounding.

### T1

*Armv8-M Floating-point Extension only*, sz == 1 UNDEFINED in single-precision only implementations.




**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sz == 0`.**

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sz == 0`.**

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sz == 1`.**

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sz == 1`.**

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(if dp_operation then ExtType_DpFp else ExtType_SpFp);
3 if VFPSmallRegisterBank() && dp_operation && ((M == '1' && op == '1') || (D == '1' && op == '
      0')) then
4     UNDEFINED;
```

```
convert_from_half = (op == '0');
lowbit = if T == '1' then 16 else 0;
if dp_operation then
    if convert_from_half then
        d = UInt(D:Vd);  m = UInt(Vm:M);
    else
        d = UInt(Vd:D);  m = UInt(M:Vm);
else
    d = UInt(Vd:D);  m = UInt(Vm:M);
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |

## Operation for all encodings

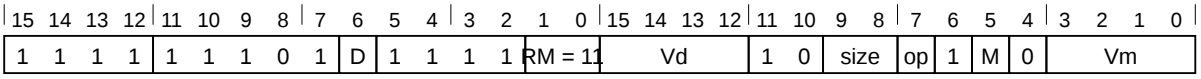
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    if convert_from_half then
        if dp_operation then
            D[d] = FPHalfToDouble(S[m]<lowbit+15:lowbit>, TRUE);
        else
            S[d] = FPHalfToSingle(S[m]<lowbit+15:lowbit>, TRUE);
    else
        if dp_operation then
            S[d]<lowbit+15:lowbit> = FPDoubleToHalf(D[m], TRUE);
        else
            S[d]<lowbit+15:lowbit> = FPSingleToHalf(S[m], TRUE);
```

## C2.4.320 VCVTM

Convert floating-point to integer with Round towards -Infinity. Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  unsigned = (op == '0');
6  round_mode = RM;
7  d = UInt(Vd:D);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

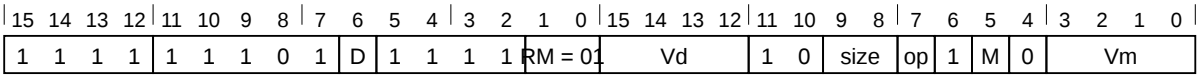
```
EncodingSpecificOperations();
ExecuteFPCheck();

case size of
    when '01'
        S[d] = FPToFixedDirected(S[m]<15:0>, 0, unsigned, round_mode, TRUE);
    when '10'
        S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);
    when '11'
        S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
```

## C2.4.321 VCVTN

Convert floating-point to integer with Round to Nearest. Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  unsigned = (op == '0');
6  round_mode = RM;
7  d = UInt(Vd:D);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

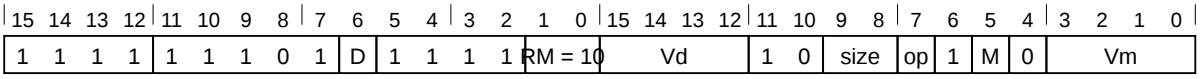
```
EncodingSpecificOperations();
ExecuteFPCheck();

case size of
    when '01'
        S[d] = FPToFixedDirected(S[m]<15:0>, 0, unsigned, round_mode, TRUE);
    when '10'
        S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);
    when '11'
        S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
```

## C2.4.322 VCVTP

Convert floating-point to integer with Round towards +Infinity. Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  unsigned = (op == '0');
6  round_mode = RM;
7  d = UInt(Vd:D);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:<br>U32 when op = 0<br>S32 when op = 1 |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

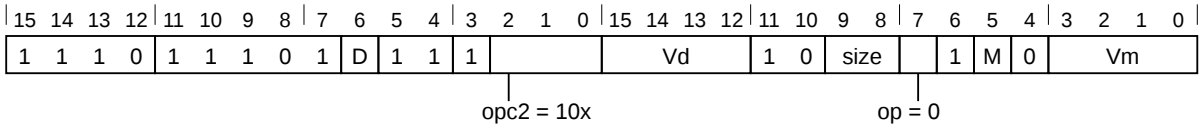
```
EncodingSpecificOperations();
ExecuteFPCheck();

case size of
    when '01'
        S[d] = FPToFixedDirected(S[m]<15:0>, 0, unsigned, round_mode, TRUE);
    when '10'
        S[d] = FPToFixedDirected(S[m], 0, unsigned, round_mode, TRUE);
    when '11'
        S[d] = FPToFixedDirected(D[m], 0, unsigned, round_mode, TRUE);
```

## C2.4.323 VCVTR

Convert floating-point to integer. Convert floating-point to integer converts a value in a register from floating-point to a 32-bit integer, using the rounding mode specified by FPSCR, and places the result in the destination register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 01`.**

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 01`.**

```
VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 10`.**

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 10`.**

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 100 && size == 11`.**

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

**Double-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `opc2 == 101 && size == 11`.**

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
to_integer = (opc2<2> == '1');
if to_integer then
    if VFPSmallRegisterBank() && dp_operation && (M == '1') then UNDEFINED;
    unsigned = (opc2<0> == '0');  round_zero = (op == '1');
    d = UInt(Vd:D);  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
else
    if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
    unsigned = (op == '0');  round_nearest = FALSE;  // FALSE selects FPSCR rounding
    m = UInt(Vm:M);  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
if size == '01' && InITBlock() then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        case size of
            when '01'
                S[d] = FPToFixed(S[m]<15:0>, 32, 0, unsigned, round_zero, TRUE);
            when '10'
                S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
            when '11'
                S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
    else
        case size of
            when '01'
                fp16 = FixedToFP(S[m], 16, 0, unsigned, round_nearest, TRUE);
                S[d] = Zeros(16):fp16;
            when '10'
                S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
            when '11'
                D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
```

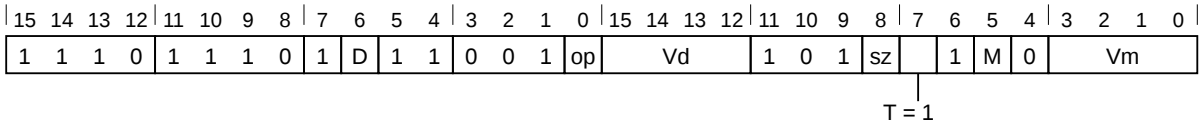
## C2.4.324 VCVTT

Floating-point Convert Top. Floating-point Convert Top does one of the following:

- Converts the half-precision value in the top half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to half-precision and writes the result into the top half of a single-precision register, preserving the other half of the target register.
- Converts the half-precision value in the top half of a single-precision register to double-precision and writes the result to a double-precision register, without intermediate rounding.
- Converts the value in the double-precision register to half-precision and writes the result into the top half of a double-precision register, preserving the other half of the target register, without intermediate rounding.

### T1

*Armv8-M Floating-point Extension only*, sz == 1 UNDEFINED in single-precision only implementations.




**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sz == 0`.**

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sz == 0`.**

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 0 && sz == 1`.**

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `op == 1 && sz == 1`.**

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(if dp_operation then ExtType_DpFp else ExtType_SpFp);
3 if VFPSmallRegisterBank() && dp_operation && ((M == '1' && op == '1') || (D == '1' && op == '
      0')) then
4     UNDEFINED;
```

```
convert_from_half = (op == '0');
lowbit = if T == '1' then 16 else 0;
if dp_operation then
    if convert_from_half then
        d = UInt(D:Vd);  m = UInt(Vm:M);
    else
        d = UInt(Vd:D);  m = UInt(M:Vm);
else
    d = UInt(Vd:D);  m = UInt(Vm:M);
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    if convert_from_half then
        if dp_operation then
            D[d] = FPHalfToDouble(S[m]<lowbit+15:lowbit>, TRUE);
        else
            S[d] = FPHalfToSingle(S[m]<lowbit+15:lowbit>, TRUE);
    else
        if dp_operation then
            S[d]<lowbit+15:lowbit> = FPDoubleToHalf(D[m], TRUE);
        else
            S[d]<lowbit+15:lowbit> = FPSingleToHalf(S[m], TRUE);
```

## C2.4.325 VDDUP, VDWDUP

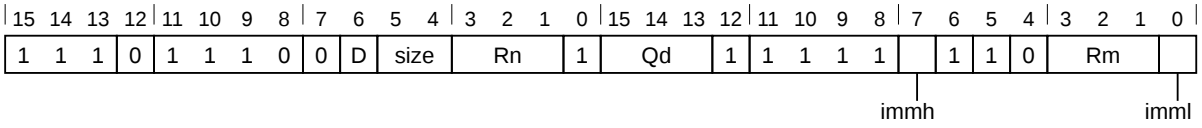
Vector Decrement and Duplicate, Vector Decrement with Wrap and Duplicate. Creates a vector with elements of successively decrementing values, starting at an offset specified by Rn. The value is decremented by the specified immediate value, which can take the following values: 1, 2, 4, 8. For the wrapping variant, if Rn and Rm are not a multiple of imm, or if Rn >= Rm, the operation is CONSTRAINED UNPREDICTABLE, with the resulting values of Rn and Qd UNKNOWN, otherwise the operation wraps so that the values written to the vector register elements are in the range [0, Rm). If Rn and Rm are not a multiple of imm, the decrementing value will not wrap. In all cases, the updated start offset is written back to Rn.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VDWDUP variant

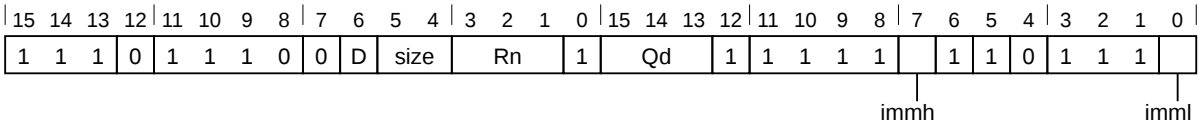
```
VDWDUP<v>.<dt> Qd, Rn, Rm, #<imm>
```

### Decode for this encoding

```
1  if Rm   == '111' then SEE "VDDUP";
2  if size == '11'  then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  if D == '1' then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(Rm:'1');
7  n        = UInt(Rn:'0');
8  wrap     = TRUE;
9  imm32    = 1 << UInt(immh:imml);
10 esize    = 8 << UInt(size);
11 elements = 32 DIV esize;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
13 if Rm == '110' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VDDUP variant

```
VDDUP<v>.<dt> Qd, Rn, #<imm>
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn:'0');
m        = integer UNKNOWN;
wrap     = FALSE;
imm32    = 1 << UInt(immh:imml);
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

<Rn> Current offset to start writing into Qd. Must be a multiple of imm. This must be an even numbered register.

### Assembler symbols for T2 encodings

<Rn> Current offset to start writing into Qd. This must be an even numbered register.

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| U8 | Encoded as | size = 00 |
| U16 | Encoded as | size = 01 |
| U32 | Encoded as | size = 10 |

<Qd> Destination vector register.

<Rm> Size of the range. Must be a multiple of imm. This must be an odd numbered register.

<imm> The increment between successive element values.
This parameter must be one of the following values:

| | | |
|---|---|---|
| #1 | Encoded as | immh = 0, imml = 0 |
| #2 | Encoded as | immh = 0, imml = 1 |
| #4 | Encoded as | immh = 1, imml = 0 |
| #8 | Encoded as | immh = 1, imml = 1 |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result    = Zeros(32);
curOffset = UInt(R[n]);
if wrap then
    bufSize = UInt(R[m]);
    if bufSize   MOD imm32 != 0 then CONSTRAINED_UNPREDICTABLE;
    if curOffset MOD imm32 != 0 then CONSTRAINED_UNPREDICTABLE;
    if curOffset >= bufSize     then CONSTRAINED_UNPREDICTABLE;
for e = 0 to elements - 1
    Elem[result, e, esize] = curOffset<esize-1:0>;
    if wrap && curOffset == 0 then
        curOffset = bufSize - imm32;
    else
        curOffset = curOffset - imm32;
```

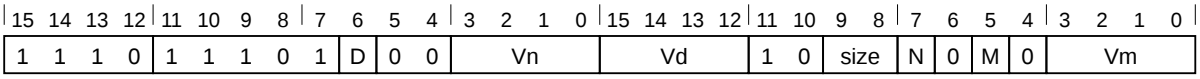
```
19  R[n] = curOffset<31:0>;
20
21  for e = 0 to 3
22      if elmtMask<e> == '1' then
23          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.326 VDIV

Floating-point Divide. Floating-point Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VDIV{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            S[d] = Zeros(16) : FPDiv(S[n]<15:0>, S[m]<15:0>, TRUE);
        when '10'
            S[d] = FPDiv(S[n], S[m], TRUE);
        when '11'
            D[d] = FPDiv(D[n], D[m], TRUE);
```

## C2.4.327 VDUP

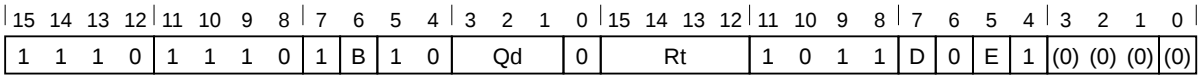
Vector Duplicate. Set each element of a vector register to the value of a general-purpose register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VDUP variant

```
VDUP<v>.<size> Qd, Rt
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if B:E == '11' then UNDEFINED;
3  if D   == '1'  then UNDEFINED;
4  d = UInt(D:Qd);
5  t = UInt(Rt);
6  case B:E of
7      when '00' esize = 32; elements = 1;
8      when '01' esize = 16; elements = 2;
9      when '10' esize =  8; elements = 4;
10     otherwise
11         // No other sizes supported
12 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
13 if Rt == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>32 Encoded as B = 0, E = 0<br>16 Encoded as B = 0, E = 1<br>8 Encoded as B = 1, E = 0 |
| <Qd> | Destination vector register. |
| <Rt> | Source general-purpose register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  for e = 0 to elements-1
8      Elem[result, e, esize] = R[t]<esize-1:0>;
9
10 for e = 0 to 3
11     if elmtMask<e> == '1' then
12         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.328 VEOR

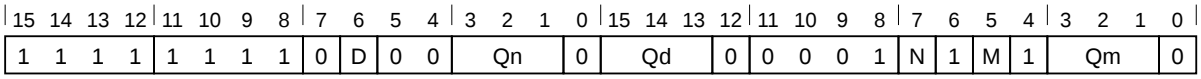
Vector Bitwise Exclusive Or. Compute a bitwise EOR of a vector register with another vector register. The result is written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VEOR variant

```
VEOR<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  n = UInt(N:Qn);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Q[n, curBeat] EOR Q[m, curBeat];
7
8  for e = 0 to 3
9      if elmtMask<e> == '1' then
10         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.329 VFMA (vector by scalar plus vector, floating-point)

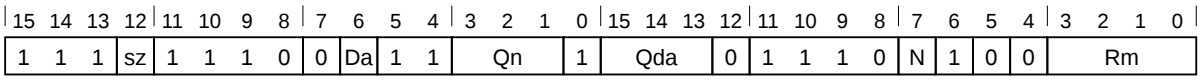
Vector Fused Multiply Accumulate. Multiply each element in a vector register by a general-purpose register value to produce a vector of results. Each result is then added to its respective element in the destination register. The result of each multiply is not rounded before the addition.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VFMA variant

```
VFMA<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if Da == '1' || N == '1' then UNDEFINED;
3  da       = UInt(Da:Qda);
4  m        = UInt(Rm);
5  n        = UInt(N:Qn);
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
9  if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qda> | Accumulator vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result   = Zeros(32);
7  op1      = Q[n, curBeat];
8  element2 = R[m]<esize-1:0>;
9  op3      = Q[da, curBeat];
10 for e = 0 to elements-1
11     element1 = Elem[op1, e, esize];
12     element3 = Elem[op3, e, esize];
13     Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FALSE);
14
```

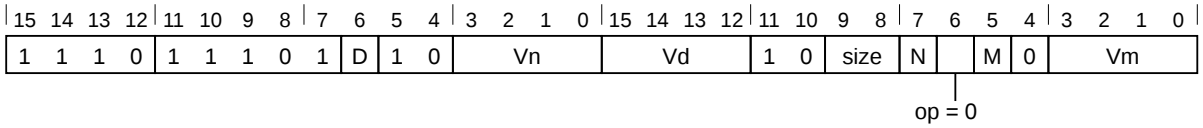
```
15  for e = 0 to 3
16      if elmtMask<e> == '1' then
17          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.330 VFMA

Floating-point Fused Multiply Accumulate. Floating-point Fused Multiply Accumulate multiplies two registers, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  op1_neg = (op == '1');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, TRUE);
        when '10'
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(S[d], op32, S[m], TRUE);
        when '11'
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
```

## C2.4.331 VFMA, VFMS (floating-point)

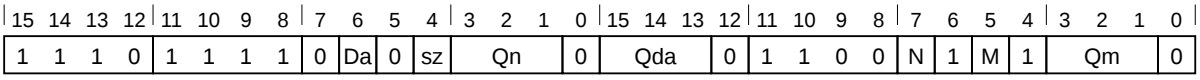
Vector Fused Multiply Accumulate, Vector Fused Multiply Subtract. Multiply each element of the first source vector register by its respective element in the second vector register. Each result is then added to or subtracted from its respective element in the destination register. The result of each multiply is not rounded before the addition or subtraction.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VFMA variant

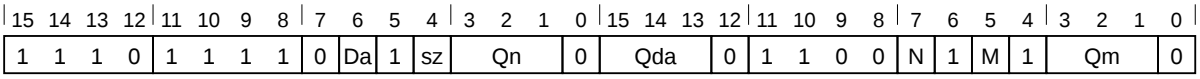
```
VFMA<v>.<dt> Qda, Qn, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if Da == '1' || M == '1' || N == '1' then UNDEFINED;
3 da       = UInt(Da:Qda);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 esize    = if sz == '1' then 16 else 32;
7 elements = 32 DIV esize;
8 add      = TRUE;
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VFMS variant

```
VFMS<v>.<dt> Qda, Qn, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if Da == '1' || M == '1' || N == '1' then UNDEFINED;
3 da       = UInt(Da:Qda);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 esize    = if sz == '1' then 16 else 32;
7 elements = 32 DIV esize;
8 add      = FALSE;
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qda> | Source and destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
op3    = Q[da, curBeat];
for e = 0 to elements-1
    element1 = Elem[op1, e, esize];
    element2 = Elem[op2, e, esize];
    element3 = Elem[op3, e, esize];
    if !add then
        element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FALSE);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.332 VFMAS (vector by vector plus scalar, floating-point)

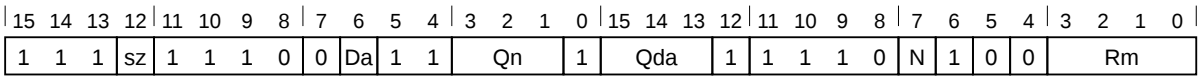
Vector Fused Multiply Accumulate Scalar. Multiply each element in the source vector by the respective element from the destination vector and add to a scalar value. The resulting values are stored in the destination vector register. The result of each multiply is not rounded before the addition.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VFMAS variant

```
VFMAS<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if Da == '1' || N == '1' then UNDEFINED;
3  da       = UInt(Da:Qda);
4  m        = UInt(Rm);
5  n        = UInt(N:Qn);
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  add      = TRUE;
9  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
10 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qda> | Source and destination vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result   = Zeros(32);
7  op1      = Q[n, curBeat];
8  op2      = Q[da, curBeat];
9  element3 = R[m]<esize-1:0>;
10 for e = 0 to elements-1
11     element1 = Elem[op1, e, esize];
12     element2 = Elem[op2, e, esize];
13     if !add then
```

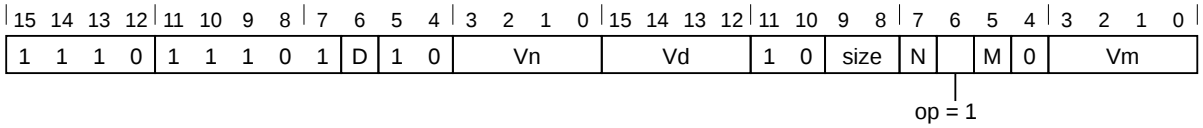
```
14          element1 = FPNeg(element1);
15      Elem[result, e, esize] = FPMulAdd(element3, element1, element2, FALSE);
16
17  for e = 0 to 3
18      if elmtMask<e> == '1' then
19          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.333 VFMS

Floating-point Fused Multiply Subtract. Floating-point Fused Multiply Subtract negates one register and multiplies it with another register, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  op1_neg = (op == '1');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

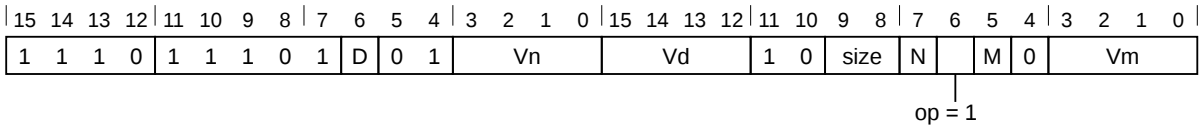
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(S[d]<15:0>, op16, S[m]<15:0>, TRUE);
        when '10'
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(S[d], op32, S[m], TRUE);
        when '11'
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
```

## C2.4.334 VFNMA

Floating-point Fused Negate Multiply Accumulate. Floating-point Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
    UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

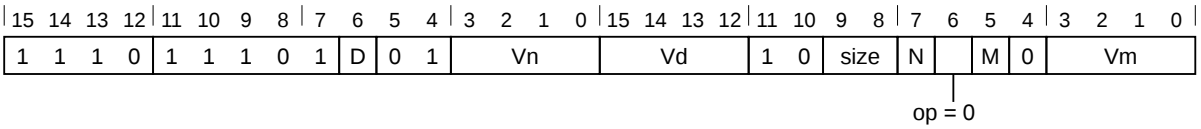
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, TRUE);
        when '10'
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
        when '11'
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
```

## C2.4.335 VFNMS

Floating-point Fused Negate Multiply Subtract. Floating-point Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  op1_neg = (op == '1');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
            S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, TRUE);
        when '10'
            op32 = if op1_neg then FPNeg(S[n]) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
        when '11'
            op64 = if op1_neg then FPNeg(D[n]) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
```

## C2.4.336 VHADD

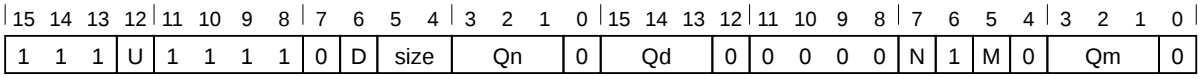
Vector Halving Add. Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is halved before being written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VHADD variant

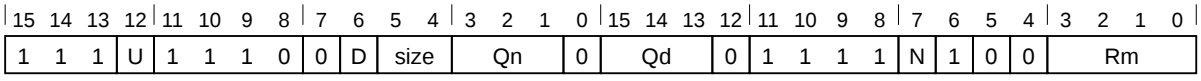
```
VHADD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VHADD variant

```
VHADD<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = TRUE;
11 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Size: indicates the size of the elements in the vector.<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if !withScalar then
    op2 = Q[m, curBeat];
for e = 0 to elements-1
    if withScalar then
        value = Int(Elem[op1, e, esize], unsigned) + Int(R[m]<esize-1:0>, unsigned);
    else
        value = Int(Elem[op1, e, esize], unsigned) + Int(Elem[op2, e, esize], unsigned);
    Elem[result, e, esize] = value<esize:1>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.337 VHCADD

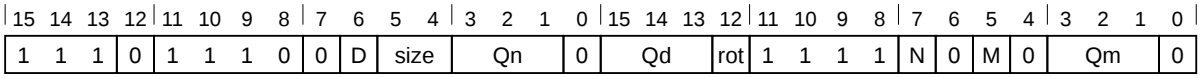
Vector Halving Complex Add with Rotate. This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, while a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components, respectively, of a complex number. The result is halved before being written to the destination register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VHCADD variant

```
VHCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  if InITBlock()                 then CONSTRAINED_UNPREDICTABLE;
10 if D:Qd == M:Qm && size == '10' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <rotate> | The rotation amount.<br>This parameter must be one of the following values:<br>#90 Encoded as rot = 0<br>#270 Encoded as rot = 1 |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
```

```
(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    case rot:curBeat<0> of
        when '00' result = (SInt(Q[n, curBeat]) - SInt(Q[m, curBeat+1]))<32:1>;
        when '01' result = (SInt(Q[n, curBeat]) + SInt(Q[m, curBeat-1]))<32:1>;
        when '10' result = (SInt(Q[n, curBeat]) + SInt(Q[m, curBeat+1]))<32:1>;
        when '11' result = (SInt(Q[n, curBeat]) - SInt(Q[m, curBeat-1]))<32:1>;
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        case rot:e<0> of
            when '00' value = SInt(Elem[op1, e, esize]) - SInt(Elem[op2, e+1, esize]);
            when '01' value = SInt(Elem[op1, e, esize]) + SInt(Elem[op2, e-1, esize]);
            when '10' value = SInt(Elem[op1, e, esize]) + SInt(Elem[op2, e+1, esize]);
            when '11' value = SInt(Elem[op1, e, esize]) - SInt(Elem[op2, e-1, esize]);
        Elem[result, e, esize] = value<esize:1>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.338 VHSUB

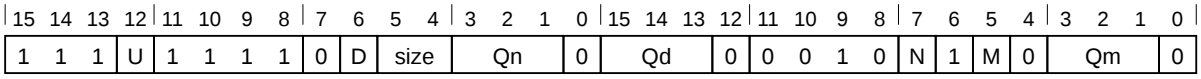
Vector Halving Subtract. Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register. The result is halved before being written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VHSUB variant

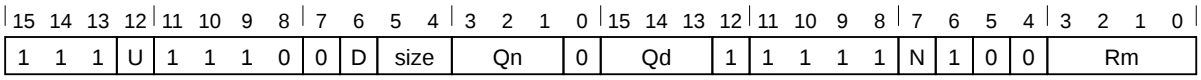
```
VHSUB<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VHSUB variant

```
VHSUB<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = TRUE;
11 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Size: indicates the size of the elements in the vector.<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if !withScalar then
    op2 = Q[m, curBeat];
for e = 0 to elements-1
    if withScalar then
        value = Int(Elem[op1, e, esize], unsigned) - Int(R[m]<esize-1:0>, unsigned);
    else
        value = Int(Elem[op1, e, esize], unsigned) - Int(Elem[op2, e, esize], unsigned);
    Elem[result, e, esize] = value<esize:1>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.339 VIDUP, VIWDUP

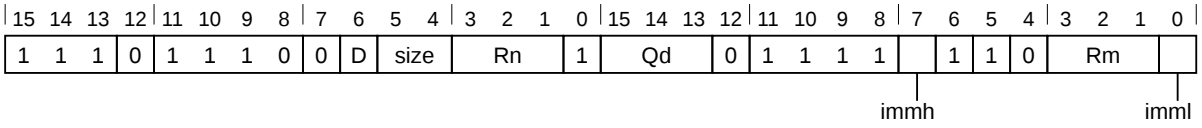
Vector Increment and Duplicate, Vector Increment with Wrap and Duplicate. Creates a vector with elements of successively incrementing values, starting at an offset specified by Rn. The value is incremented by the specified immediate value, which can take the following values: 1, 2, 4, 8. For the wrapping variant, if Rn and Rm are not a multiple of imm, or if Rn >= Rm, the operation is CONSTRAINED UNPREDICTABLE, with the resulting values of Rn and Qd UNKNOWN, otherwise the operation wraps so that the values written to the vector register elements are in the range [0, Rm). If Rn and Rm are not a multiple of imm, the incrementing value will not wrap. In all cases, the updated start offset is written back to Rn.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VIWDUP variant

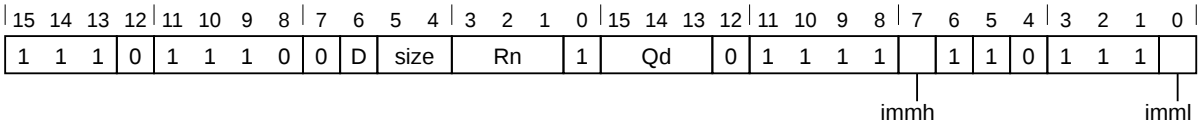
```
VIWDUP<v>.<dt> Qd, Rn, Rm, #<imm>
```

### Decode for this encoding

```
1  if Rm   == '111' then SEE "VIDUP";
2  if size == '11'  then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  if D == '1' then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(Rm:'1');
7  n        = UInt(Rn:'0');
8  wrap     = TRUE;
9  imm32    = 1 << UInt(immh:imml);
10 esize    = 8 << UInt(size);
11 elements = 32 DIV esize;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
13 if Rm == '110' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VIDUP variant

```
VIDUP<v>.<dt> Qd, Rn, #<imm>
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn:'0');
m        = integer UNKNOWN;
wrap     = FALSE;
imm32    = 1 << UInt(immh:imml);
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

<Rn> Current offset to start writing into Qd. Must be a multiple of imm. This must be an even numbered register.

### Assembler symbols for T2 encodings

<Rn> Current offset to start writing into Qd. This must be an even numbered register.

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| U8 | Encoded as | size = 00 |
| U16 | Encoded as | size = 01 |
| U32 | Encoded as | size = 10 |

<Qd> Destination vector register.

<Rm> Size of the range. Must be a multiple of imm. This must be an odd numbered register.

<imm> The increment between successive element values.
This parameter must be one of the following values:

| | | |
|---|---|---|
| #1 | Encoded as | immh = 0, imml = 0 |
| #2 | Encoded as | immh = 0, imml = 1 |
| #4 | Encoded as | immh = 1, imml = 0 |
| #8 | Encoded as | immh = 1, imml = 1 |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result    = Zeros(32);
curOffset = UInt(R[n]);
if wrap then
    bufSize = UInt(R[m]);
    if bufSize   MOD imm32 != 0 then CONSTRAINED_UNPREDICTABLE;
    if curOffset MOD imm32 != 0 then CONSTRAINED_UNPREDICTABLE;
    if curOffset >= bufSize     then CONSTRAINED_UNPREDICTABLE;
for e = 0 to elements - 1
    Elem[result, e, esize] = curOffset<esize-1:0>;
    curOffset = curOffset + imm32;
    if wrap && curOffset == bufSize then
        curOffset = 0;
R[n] = curOffset<31:0>;
```

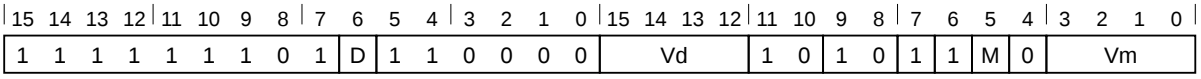
```
19
20 for e = 0 to 3
21     if elmtMask<e> == '1' then
22         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.340 VINS

Floating-point move Insertion. Floating-point move Insertion copies the lower 16 bits of the 32-bit source Floating-point Extension register into the upper 16 bits of the 32-bit destination Floating-point Extension register, while preserving the values in the remaining bits.

### T1

*Armv8.1-M Floating-point Extension only*




#### T1 variant

```
VINS<q>.F16 <Sd>, <Sm>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_HpFp);
2 if InITBlock() then UNPREDICTABLE;
3 d = UInt(Vd:D); m = UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteFPCheck();
4     S[d]<31:16> = S[m]<15:0>;
```

## C2.4.341 VLD2

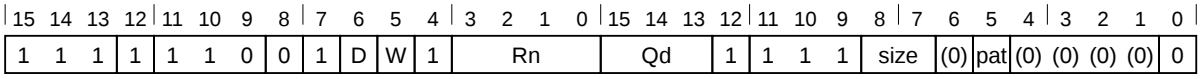
Vector Deinterleaving Load - Stride 2. Loads two 64-bit contiguous blocks of data from memory and writes them to parts of 2 destination registers. The parts of the destination registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 2 times with the same base address and destination registers, but with different pat values, the effect is to load data from memory and to deinterleave it into the specified registers with a stride of 2. The base address register can optionally be incremented by 32.

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VLD2 variant (Non writeback: W=0)

```
VLD2<pat>.<size> {Qd, Qd+1}, [Rn]
```

#### T1: VLD2 variant (Writeback: W=1)

```
VLD2<pat>.<size> {Qd, Qd+1}, [Rn]!
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  pattern  = UInt(pat);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  wback    = (W == '1');
10 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
11 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
12 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
13 if UInt(D:Qd) > 6           then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <pat> | Specifies the pattern of register elements and memory addresses to access.<br>This parameter must be one of the following values:<br>0 Encoded as pat = 0<br>1 Encoded as pat = 1 |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as size = 00<br>16 Encoded as size = 01<br>32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Rn> | The base register for the target address. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, -) = GetCurInstrBeat();

// Pre-calculate variables for memory / register access patterns
addrWordOffset = curBeat<1> : (UInt(curBeat<1>) + pattern)<0> : curBeat<0>;
baseAddress    = R[n] + ZeroExtend(addrWordOffset:'00', 32);
xBeat          = UInt(curBeat<1> : (pattern<0> EOR curBeat<1>));

for e = 0 to elements-1
    address = baseAddress + (e * (esize DIV 8));
    case esize of
        when 8
            y  = UInt(e<0>);
            xE = UInt(curBeat<0> : e<1>);
        when 16
            y  = UInt(e<0>);
            xE = UInt(curBeat<0>);
        when 32
            y  = UInt(curBeat<0>);
            xE = 0;
    Elem[Q[d + y, xBeat], xE, esize] = MemA_MVE[address, esize DIV 8];

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = R[n] + 32;
```

## C2.4.342 VLD4

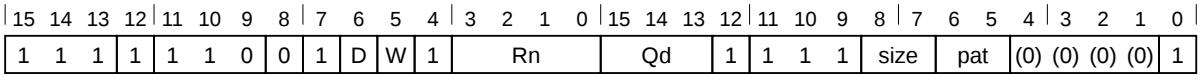
Vector Deinterleaving Load - Stride 4. Loads two 64-bit contiguous blocks of data from memory and writes them to parts of 4 destination registers. The parts of the destination registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 4 times with the same base address and destination registers, but with different pat values, the effect is to load data from memory and to deinterleave it into the specified registers with a stride of 4. The base address register can optionally be incremented by 64.

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VLD4 variant (Non writeback: W=0)

```
VLD4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]
```

#### T1: VLD4 variant (Writeback: W=1)

```
VLD4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]!
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  pattern  = UInt(pat);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  wback    = (W == '1');
10 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
11 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
12 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
13 if UInt(D:Qd) > 4           then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <pat> | Specifies the pattern of register elements and memory addresses to access.<br>This parameter must be one of the following values:<br>0 Encoded as pat = 00<br>1 Encoded as pat = 01<br>2 Encoded as pat = 10<br>3 Encoded as pat = 11 |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as size = 00<br>16 Encoded as size = 01<br>32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Rn> | The base register for the target address. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, -) = GetCurInstrBeat();

// Pre-calculate variables for memory / register access patterns
addrWordOffset = curBeat<1> : (UInt(curBeat<1>) + pattern)<1:0> : curBeat<0>;
baseAddress    = R[n] + ZeroExtend(addrWordOffset:'00', 32);
xBeat          = UInt(curBeat<1> : (pattern<1> EOR (pattern<0> AND curBeat<1>)));

for e = 0 to elements-1
    address = baseAddress + (e * (esize DIV 8));
    case esize of
        when 8
            y  = UInt(e<1:0>);
            xE = UInt((pattern<0> EOR curBeat<1>) : curBeat<0>);
        when 16
            y  = UInt(curBeat<0> : e<0>);
            xE = UInt(pattern<0> EOR curBeat<1>);
        when 32
            y  = UInt((pattern<0> EOR curBeat<1>) : curBeat<0>);
            xE = 0;
    Elem[Q[d + y, xBeat], xE, esize] = MemA_MVE[address, esize DIV 8];

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = R[n] + 64;
```

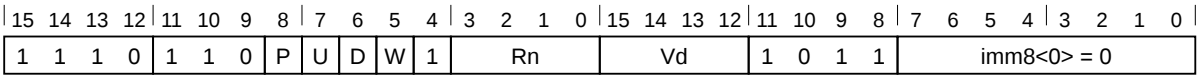
## C2.4.343 VLDM

Floating-point Load Multiple. Floating-point Load Multiple loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

This instruction is used by the alias VPOP.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1 && Rn != 1111`.**

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

### Decode for this encoding

```
1  if P == '0' && U == '0' && W == '0'                then SEE "Related encodings";
2  if P == '0' && U == '1' && W == '0' && Rn == '1111' then SEE "VSCCLRM";
3  if P == '1' && W == '0'                            then SEE VLDR;
4  CheckDecodeFaults(ExtType_MveOrFp);
5  if P == U && W == '1' then UNDEFINED;
6  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
7  single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
8  d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
9  regs = UInt(imm8) DIV 2;
10 if n == 15 then UNPREDICTABLE;
11 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

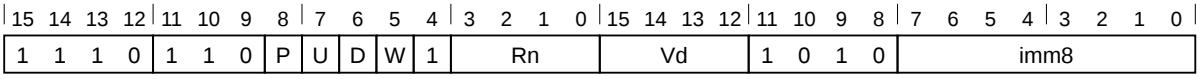
#### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

### T2

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1 && Rn != 1111`.**

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

## Decode for this encoding

```
1  if P == '0' && U == '0'                              then SEE "Related encodings";
2  if P == '0' && U == '1' && W == '0' && Rn == '1111' then SEE "VSCCLRM";
3  if P == '1' && W == '0'                              then SEE VLDR;
4  CheckDecodeFaults(ExtType_MveOrFp);
5  if P == '1' && U == '1' && W == '1' then UNDEFINED;
6  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
7  single_regs = TRUE;  add = (U == '1');  wback = (W == '1');
8  d = UInt(Vd:D);  n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
9  regs   = UInt(imm8);
10 topReg = d+regs-1;
11 if n == 15                          then UNPREDICTABLE;
12 if regs == 0        || topReg > 63 then UNPREDICTABLE;
13 if topReg<0> == '0' && topReg > 31 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a VLDM with the same addressing mode but loads no registers.

### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the floating-point registers are UNKNOWN. If the instruction specifies writeback, the base register becomes UNKNOWN. This behavior does not affect any general-purpose registers.

## Alias conditions

| Alias | preferred when |
|---|---|
| VPOP | `P == '0' && U == '1' && W == '1' && Rn == '1101'` |

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0. |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register. |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

## Operation for all encodings

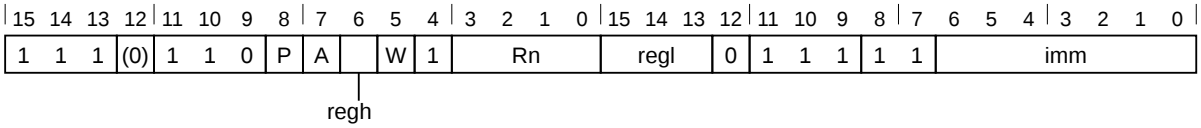
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    address = if add then R[n]       else R[n]-imm32;
    regval  = if add then R[n]+imm32 else R[n]-imm32;

    // Determine if the stack pointer limit must be checked
    if n == 13 && wback then
        // If memory operation is not performed as a result of a stack limit violation,
        // and the write-back of the SP itself does not raise a stack limit violation, it
        // is "IMPLEMENTATION_DEFINED" whether a SPLIM exception is raised.
        // Arm recommends that any instruction which discards a memory access as
        // a result of a stack limit violation, and where the write-back of the SP itself
        // does not raise a stack limit violation, generates an SPLIM exception.
        if boolean IMPLEMENTATION_DEFINED "SPLIM exception on invalid memory access" then
            if ViolatesSPLim(LookUpSP(), address) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to
                    HardFault
                excInfo = CreateException(UsageFault);
                HandleException(excInfo);
        applylimit = TRUE;
    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !(applylimit && ViolatesSPLim(LookUpSP(), regval)) then
        for r = 0 to regs-1
            if single_regs then
                S[d+r]  = MemA[address,4];
                address = address+4;
            else
                if (d+r) < 16 || !VFPSmallRegisterBank() then
                    word1  = MemA[address,4];  word2 = MemA[address+4,4];
                    // Combine the word-aligned words in the correct order for
                    // current endianness.
                    D[d+r] = if BigEndian(address, 8) then word1:word2 else word2:word1;
                elsif boolean UNKNOWN then
                    - = MemA[address,4]; - = MemA[address+4,4];
                address = address+8;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

## C2.4.344 VLDR (System Register)

Load System Register. Load a system register from memory. The target address is calculated from a base register plus an immediate offset. If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0. This instruction is UNDEFINED if executed in Non-secure state

### T1

*Armv8.1-M Floating-point Extension and / or Armv8.1-M MVE*




#### T1: VLDR variant (Offset: P=1, W=0)

```
VLDR<c> <reg>, [Rn{, #+/-<imm>}]
```

#### T1: VLDR variant (Pre-indexed: P=1, W=1)

```
VLDR<c> <reg>, [Rn, #+/-<imm>]!
```

#### T1: VLDR variant (Post-indexed: P=0, W=1)

```
VLDR<c> <reg>, [Rn], #+/-<imm>
```

### Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  if !HasArchVersion(Armv8p1) then
3      UFSR.NOCP = '1';
4      HandleException(CreateException(UsageFault));
5  fpCxtAccess    = (regh:regl == '111x');
6  fpCxtNSSAccess = (regh:regl == '1110');
7  fpInactive     = !HaveMveOrFPExt() || (FPCCR_NS.ASPEN == '1' && CONTROL.FPCA == '0');
8  if fpCxtAccess then
9      if !HaveMainExt() || !IsSecure() then UNDEFINED;
10     if !fpInactive then
11         HandleException(CheckCPEnabled(10));
12 else
13     CheckDecodeFaults(ExtType_MveOrFp);
14 n     = UInt(Rn);
15 index = (P == '1');
16 add   = (A == '1');
17 wback = (W == '1');
18 r     = regh:regl;
19 imm32 = ZeroExtend(imm:'00', 32);
20 if (regh:regl) IN {'10xx', '01xx', '0011', '0000'} then CONSTRAINED_UNPREDICTABLE;
21 if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <reg> | The system register to access<br>This parameter must be one of the following values:<br>FPSCR Encoded as regh = 0, regl = 001<br>FPSCR_nzcvqc Encoded as regh = 0, regl = 010<br>VPR Encoded as regh = 1, regl = 100<br>P0 Encoded as regh = 1, regl = 101 |

FPCXTNS Encoded as regh = 1, regl = 110
FPCXTS Encoded as regh = 1, regl = 111

<Rn> The base register for the target address.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

## Operation for all encodings

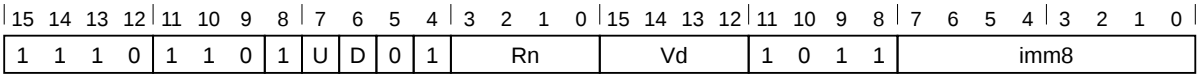
```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !fpCxtNSSAccess then
        ExecuteFPCheck();
    elsif !fpInactive then
        PreserveFPState();
    SerializeVFP();
    VFPExcBarrier();

    offsetAddr = if add   then (R[n] + imm32) else (R[n] - imm32);
    address    = if index then offsetAddr     else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offsetAddr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        case r of
            when '0001'
                FPSCR = MemA[address, 4];
            when '0010'
                // Only update the N, Z, C, V, and QC flags
                FPSCR<31:27> = MemA[address, 4]<31:27>;
            when '1100'
                if HaveMve() then
                    if CurrentModeIsPrivileged() then
                        VPR = MemA[address, 4];
                else
                    UNPREDICTABLE;
            when '1101'
                if HaveMve() then
                    VPR.P0 = MemA[address, 4]<15:0>;
                else
                    UNPREDICTABLE;
            when '1110'
                if (HaveFPExt() || HaveMve()) && !fpInactive then
                    FPCXT_Type cxt = MemA[address, 4];
                    CONTROL_S.SFPA = cxt.SFPA;
                    FPSCR          = Zeros(4):cxt<27:0>;
            when '1111'
                FPCXT_Type cxt = MemA[address, 4];
                CONTROL_S.SFPA = cxt.SFPA;
                FPSCR          = Zeros(4):cxt<27:0>;
            otherwise
                UNPREDICTABLE;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then
        RSPCheck[n] = offsetAddr;
```

## C2.4.345 VLDR

Floating-point Load Register. Floating-point Load Register loads a Floating-point Extension register from memory, using an address from a general-purpose register, with an optional offset.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Literal variant**

**Applies when `Rn == 1111`.**

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

**Offset variant**

**Applies when `Rn != 1111`.**

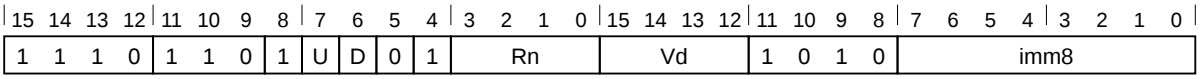
```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveOrFp);
2 if VFPSmallRegisterBank() && (D == '1') then UNDEFINED;
3 fp_size = 64;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
4 d = UInt(D:Vd);  n = UInt(Rn);
```

### T2

*Armv8-M Floating-point Extension only or MVE*




**Literal variant**

**Applies when `Rn == 1111`.**

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

**Offset variant**

**Applies when `Rn != 1111`.**

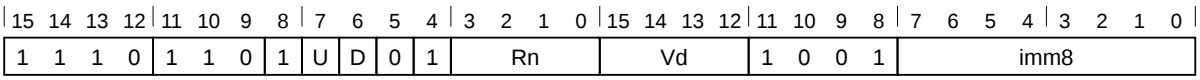
```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveOrFp);
2 fp_size = 32;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
3 d = UInt(Vd:D);  n = UInt(Rn);
```

### T3

*Armv8.1-M Floating-point Extension only or MVE*




**Literal variant**

**Applies when `Rn == 1111`.**

```
VLDR{<c>}{<q>}.16 <Sd>, <label>
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

**Offset variant**

**Applies when `Rn != 1111`.**

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

## Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveOrFp);
2 fp_size = 16;  add = (U == '1');  imm32 = ZeroExtend(imm8:'0', 32);
3 d = UInt(Vd:D);  n = UInt(Rn);
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| .64 | Optional data size specifiers. |
| <Dd> | The destination register for a doubleword load. |
| .32 | Optional data size specifiers. |
| <Sd> | The destination register for a singleword load. |
| <label> | The label of the literal data item to be loaded. The assembler calculates the required value of the offset from the Align(PC, 4) value of the instruction to this label. Permitted values are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE. If the offset is negative, imm32 is equal to minus the offset and add == FALSE. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | The immediate offset used for forming the address. For the immediate forms of the syntax, <imm> can be omitted, in which case the #0 form of the instruction is assembled. Permitted values are multiples of 4 in the range 0 to 1020. |

## Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4      base = if n == 15 then Align(PC,4) else R[n];
5      address = if add then (base + imm32) else (base - imm32);
6      case fp_size of
7          when 16
8              S[d] = Zeros(16) : MemA[address,2];
9          when 32
10             S[d] = MemA[address,4];
11         when 64
12             word1 = MemA[address,4];  word2 = MemA[address+4,4];
```

```
13              // Combine the word-aligned words in the correct order for current endianness.
14              D[d] = if BigEndian(address, 8) then word1:word2 else word2:word1;
```

## C2.4.346 VLDRB, VLDRH, VLDRW

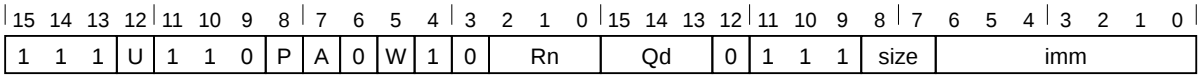
Vector Load Register. Load consecutive elements from memory into a destination vector register. Each element loaded will be the zero or sign-extended representation of the value in memory. In indexed mode, the target address is calculated from a base register offset by an immediate value. Otherwise, the base register address is used directly. The sum of the base register and the immediate value can optionally be written back to the base register. Predicated lanes are zeroed instead of retaining their previous values.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VLDRB variant (Offset: P=1, W=0)

```
VLDRB<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

#### T1: VLDRB variant (Pre-indexed: P=1, W=1)

```
VLDRB<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

#### T1: VLDRB variant (Post-indexed: P=0, W=1)

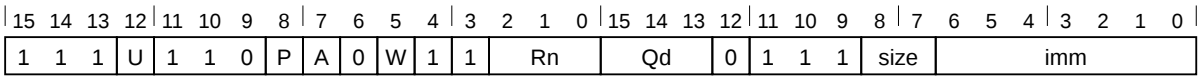
```
VLDRB<v>.<dt> Qd, [Rn], #+/-<imm>
```

### Decode for this encoding

```
if P    == '0' && W == '0' then SEE "Related encodings";
if size == '11'            then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if size == '00' then UNDEFINED;
d        = UInt(Qd);
n        = UInt(Rn);
msize    = 8;
mbytes   = msize DIV 8;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
imm32    = ZeroExtend(imm, 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = (U == '1');
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VLDRH variant (Offset: P=1, W=0)

```
VLDRH<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

**T2: VLDRH variant (Pre-indexed: P=1, W=1)**

```
VLDRH<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

**T2: VLDRH variant (Post-indexed: P=0, W=1)**

```
VLDRH<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P    == '0' && W == '0' then SEE "Related encodings";
if size == '11'            then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if size == '00' then UNDEFINED;
d        = UInt(Qd);
n        = UInt(Rn);
msize    = 16;
mbytes   = msize DIV 8;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
imm32    = ZeroExtend(imm:'0', 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = (U == '1');
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T5

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 1 | Rn | | | | Qd | | | | 1 | 1 | 1 | 1 | 0 | 0 | imm | | | | | |

**T5: VLDRB variant (Offset: P=1, W=0)**

```
VLDRB<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

**T5: VLDRB variant (Pre-indexed: P=1, W=1)**

```
VLDRB<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

**T5: VLDRB variant (Post-indexed: P=0, W=1)**

```
VLDRB<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn);
msize    = 8;
mbytes   = msize DIV 8;
esize    = msize;
elements = 32 DIV esize;
imm32    = ZeroExtend(imm, 32);
index    = (P == '1');
add      = (A == '1');
```

```
wback    = (W == '1');
unsigned = TRUE;
if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 1 | Rn | | | | Qd | | | | 1 | 1 | 1 | 1 | 0 | 1 | imm | | | | | |

### T6: VLDRH variant (Offset: P=1, W=0)

```
VLDRH<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T6: VLDRH variant (Pre-indexed: P=1, W=1)

```
VLDRH<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T6: VLDRH variant (Post-indexed: P=0, W=1)

```
VLDRH<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn);
msize    = 16;
mbytes   = msize DIV 8;
esize    = msize;
elements = 32 DIV esize;
imm32    = ZeroExtend(imm:'0', 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = TRUE;
if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
```

## T7

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 1 | Rn | | | | Qd | | | | 1 | 1 | 1 | 1 | 1 | 0 | imm | | | | | |

### T7: VLDRW variant (Offset: P=1, W=0)

```
VLDRW<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T7: VLDRW variant (Pre-indexed: P=1, W=1)

```
VLDRW<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T7: VLDRW variant (Post-indexed: P=0, W=1)

```
VLDRW<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  msize    = 32;
7  mbytes   = msize DIV 8;
8  esize    = msize;
9  elements = 32 DIV esize;
10 imm32    = ZeroExtend(imm:'00', 32);
11 index    = (P == '1');
12 add      = (A == '1');
13 wback    = (W == '1');
14 unsigned = TRUE;
15 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
16 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
17 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

<imm> The signed immediate value that is added to base register to calculate the target address.

## Assembler symbols for T2 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 2.

## Assembler symbols for T5 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address.

## Assembler symbols for T6 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 2.

## Assembler symbols for T7 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

## Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Qd> Destination vector register.

<Rn> The base register for the target address.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result     = Zeros(32);
offsetAddr = if add   then (R[n] + imm32) else (R[n] - imm32);
address    = if index then offsetAddr     else R[n];
address    = address + (curBeat * mbytes * elements);

for e = 0 to elements-1
    if elmtMask<e*(esize >> 3)> == '1' then
        Elem[result, e, esize] = Extend(MemA_MVE[address + (e * mbytes), mbytes], unsigned);

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = offsetAddr;

Q[d, curBeat] = result;
```

## C2.4.347 VLDRB, VLDRH, VLDRW, VLDRD (vector)

Vector Gather Load. Load a byte, halfword, word, or doubleword from memory at the address contained in either:

a) A base register R[n] plus an offset contained in each element of Q[m], optionally shifted by the element size, or

b) Each element of Q[m] plus an immediate offset. The base element can optionally be written back, irrespective of predication, with that value incremented by the immediate or by the immediate scaled by the memory element size.

Each element loaded will be the zero or sign-extended representation of the value in memory. The result is written back into the corresponding element in the destination vector register Q[d]. Predicated lanes are zeroed instead of retaining their previous values.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 0 | 0 | 1 | D | 0 | 1 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 0 | M | 0 | Qm | | | | os |

#### T1: VLDRB variant

```
VLDRB<v>.<dt> Qd, [Rn, Qm]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D     == '1' || M == '1'                    then UNDEFINED;
3  if size == '11' || (U == '0' && size == '00') then UNDEFINED;
4  d           = UInt(D:Qd);
5  m           = UInt(M:Qm);
6  n           = UInt(Rn);
7  msize       = '00';
8  esize       = 8 << UInt(size);
9  elements    = 32 DIV esize;
10 mesize      = 8;
11 add         = TRUE;
12 useReg      = TRUE;
13 scaleOffset = (os == '1');
14 unsigned    = (U  == '1');
15 wback       = FALSE;
16 if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
17 if D:Qd == M:Qm   then CONSTRAINED_UNPREDICTABLE;
18 if Rn   == '1111' then CONSTRAINED_UNPREDICTABLE;
19 if os   == '1'    then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 0 | 0 | 1 | D | 0 | 1 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 0 | M | 1 | Qm | | | | os |

#### T2: VLDRH variant

```
VLDRH<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D     == '1' || M == '1' then UNDEFINED;
if size == '11' || size == '00' || (U == '0' && size == '01') then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '01';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 16;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
unsigned    = (U  == '1');
wback       = FALSE;
if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
if D:Qd == M:Qm   then CONSTRAINED_UNPREDICTABLE;
if Rn   == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 0 | 0 | 1 | D | 0 | 1 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 1 | M | 0 | Qm | | | | os |

### T3: VLDRW variant

```
VLDRW<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1'     then UNDEFINED;
if U == '0' || size != '10' then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '10';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 32;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
unsigned    = (U  == '1');
wback       = FALSE;
if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
if D:Qd == M:Qm   then CONSTRAINED_UNPREDICTABLE;
if Rn   == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 0 | 0 | 1 | D | 0 | 1 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 1 | M | 1 | Qm | | | | os |

### T4: VLDRD variant

```
VLDRD<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1'     then UNDEFINED;
if U == '0' || size != '11' then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '11';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 64;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
unsigned    = (U  == '1');
wback       = FALSE;
if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
if D:Qd == M:Qm   then CONSTRAINED_UNPREDICTABLE;
if Rn   == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T5

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | A | D | W | 1 | Qm | | | | (0) | Qd | | | 1 | 1 | 1 | 1 | 0 | M | imm | | | | | |

### T5: VLDRW variant (Non writeback: W=0)

```
VLDRW<v>.<dt> Qd, [Qm{, #+/-<imm>}]
```

### T5: VLDRW variant (Writeback: W=1)

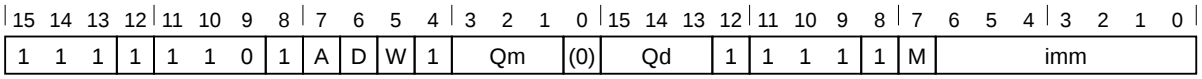
```
VLDRW<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = integer UNKNOWN;
size        = '10';
msize       = size;
offset      = ZeroExtend(imm:Zeros(UInt(size)), 32);
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 32;
add         = (A == '1');
useReg      = FALSE;
scaleOffset = FALSE;
unsigned    = TRUE;
wback       = (W == '1');
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if D:Qd == M:Qm then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*




**T6: VLDRD variant (Non writeback: W=0)**

```
VLDRD<v>.<dt> Qd, [Qm{, #+/-<imm>}]
```

**T6: VLDRD variant (Writeback: W=1)**

```
VLDRD<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

## Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d           = UInt(D:Qd);
4  m           = UInt(M:Qm);
5  n           = integer UNKNOWN;
6  size        = '11';
7  msize       = size;
8  offset      = ZeroExtend(imm:Zeros(UInt(size)), 32);
9  esize       = 8 << UInt(size);
10 elements    = 32 DIV esize;
11 mesize      = 64;
12 add         = (A == '1');
13 useReg      = FALSE;
14 scaleOffset = FALSE;
15 unsigned    = TRUE;
16 wback       = (W == '1');
17 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
18 if D:Qd == M:Qm then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned. Operations that do not perform widening are always unsigned (encoded with U=1), the equivalent sized floating and signless datatypes are allowed but are an alias for the unsigned version.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| U8 | Encoded as | size = 00, U = 1 |
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

## Assembler symbols for T2 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned. Operations that do not perform widening are always unsigned (encoded with U=1), the equivalent sized floating and signless datatypes are allowed but are an alias for the unsigned version.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

U16 Encoded as size = 01, U = 1
S32 Encoded as size = 10, U = 0
U32 Encoded as size = 10, U = 1

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

### Assembler symbols for T3 encodings

<dt> Unsigned flag: S indicates signed, U indicates unsigned. Operations that do not perform widening are always unsigned (encoded with U=1), the equivalent sized floating and signless datatypes are allowed but are an alias for the unsigned version.
This parameter must be the following value:
U32 Encoded as size = 10, U = 1

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

### Assembler symbols for T4 encodings

<dt> Unsigned flag: S indicates signed, U indicates unsigned. Operations that do not perform widening are always unsigned (encoded with U=1), the equivalent sized floating and signless datatypes are allowed but are an alias for the unsigned version.
This parameter must be the following value:
U64 Encoded as size = 11, U = 1

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

### Assembler symbols for T5 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<Qm> The base register for the target address.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

### Assembler symbols for T6 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<Qm> The base register for the target address.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 8.

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<Qd> Destination vector register.

<Rn> The base register for the target address.

<os> The amount by which the vector offset is left shifted by before being added to the general-purpose base address. If the value is present it must correspond to memory transfer size (1=half word, 2=word, 3=double word).
This parameter must be one of the following values:

<omitted> Encoded as os = 0
<Offset scaled> Encoded as os = 1

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
if esize == 64 then
    // 64 bit accesses read their base address or offset from the first element
    // in each pair of 32 bit elements.
    if useReg then
        baseAddr = R[n];
        offset   = Q[m, UInt(curBeat<1>:'0')];
        if scaleOffset then
            offset = LSL(offset, UInt(msize));
    else
        baseAddr = Q[m, UInt(curBeat<1>:'0')];
    offsetAddress = if add then baseAddr + offset else baseAddr - offset;
    bigEndian     = BigEndian(offsetAddress, 8);
    address       = (if (curBeat<0> == '0') == bigEndian then offsetAddress + 4
                                                         else offsetAddress);
    if elmtMask<0> == '1' then
        result = MemA_MVE[address, 4];
    // Address writeback is not predicated
    if wback && (curBeat<0> == '1') then
        Q[m, curBeat-1] = offsetAddress<31:0>;
else
    // 32, 16, or 8 bit accesses
    for e = 0 to (elements - 1)
        if useReg then
            baseAddr = R[n];
            offset   = ZeroExtend(Elem[Q[m, curBeat], e, esize], 32);
            if scaleOffset then
                offset = LSL(offset, UInt(msize));
        else
            // 16 / 8 bit vector+immediate accesses are not supported
            baseAddr = Q[m, curBeat];
        address = if add then baseAddr + offset else baseAddr - offset;
        if elmtMask<e*(esize>>3)> == '1' then
            memValue               = MemA_MVE[address, mesize DIV 8];
            Elem[result, e, esize] = Extend(memValue, esize, unsigned);
        // Address writeback is not predicated
        if wback then
            Elem[Q[m, curBeat], e, esize] = address<esize-1:0>;


Q[d, curBeat] = result;
```

## C2.4.348 VLLDM

Floating-point Lazy Load Multiple. Floating-point Lazy Load Multiple restores the contents of the Secure floating-point registers that were protected by a VLSTM instruction, and marks the floating-point context as active.

If the lazy state preservation set up by a previous VLSTM instruction is active (FPCCR.LSPACT == 1), this instruction deactivates lazy state preservation and enables access to the Secure floating-point registers.

If lazy state preservation is inactive (FPCCR.LSPACT == 0), either because lazy state preservation was not enabled (FPCCR.LSPEN == 0) or because a floating-point instruction caused the Secure floating-point register contents to be stored to memory, this instruction loads the stored Secure floating-point register contents back into the floating-point registers.

If Secure floating-point is not in use (CONTROL_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

### T1

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | (0) | 1 | 1 | Rn | | | | (0) | (0) | (0) | (0) | 1 | 0 | 1 | 0 | 0 | (0) | (0) | (0) | (0) | (0) | (0) | (0) |

#### T1 variant

```
VLLDM{<c>}{<q>} <Rn> {, <reglist>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  if !IsSecure() || !VFPSmallRegisterBank() then UNDEFINED;
4  lowRegsOnly = TRUE;
5  if n == 15 then UNPREDICTABLE;
```

### T2

*Armv8.1-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | (0) | 1 | 1 | Rn | | | | (0) | (0) | (0) | (0) | 1 | 0 | 1 | 0 | 1 | (0) | (0) | (0) | (0) | (0) | (0) | (0) |

#### T2 variant

```
VLLDM{<c>}{<q>} <Rn>, <reglist>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  if !IsSecure() then UNDEFINED;
4  lowRegsOnly = FALSE;
5  if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| <reglist> | For encoding T1: Optional register list of {D0-D15}, having the register list is the preferred disassembly<br>For encoding T2: Mandatory register list of {D0-D31} |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if CONTROL_S.SFPA == '1' then
        // Check access to the co-processor is permitted
        exc = CheckCPEnabled(10);
        HandleException(exc);

        if FPCCR_S.LSPACT == '1' then // state in FP is still valid
            FPCCR_S.LSPACT = '0';
        else
            if !IsAligned(R[n],8) then
               UFSR.UNALIGNED = '1';
               exc = CreateException(UsageFault);
               HandleException(exc);

            for i = 0 to 15
                S[i] = MemA[R[n] + (4*i), 4];
            FPSCR = MemA[R[n] + 0x40, 4];
            if HaveMve() then
                VPR = MemA[R[n] + 0x44, 4];
            if FPCCR_S.TS == '1' then
                for i = 0 to 15
                    S[i+16] = MemA[R[n] + 0x48 + (4*i), 4];
            if !lowRegsOnly && boolean UNKNOWN then
                for i = 0 to 31
                    - = MemA[R[n] + 0x88 + (4*i), 4];

        CONTROL.FPCA = '1';
```

## C2.4.349 VLSTM

Floating-point Lazy Store Multiple. Floating-point Lazy Store Multiple stores the contents of Secure floating-point registers to a prepared stack frame, and clears the Secure floating-point registers.

If floating-point lazy preservation is enabled (FPCCR.LSPEN == 1), then the next time a floating-point instruction other than VLSTM or VLLDM is executed:

- The contents of Secure floating-point registers are stored to memory.
- The Secure floating-point registers are cleared.

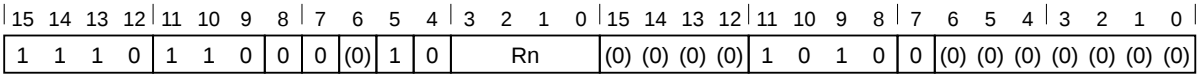
If Secure floating-point is not in use (CONTROL_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

### T1

*Armv8-M Main Extension only*




#### T1 variant

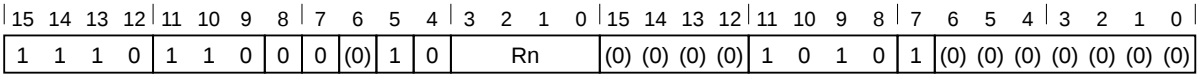
```
VLSTM{<c>}{<q>} <Rn> {, <reglist>}
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  if !IsSecure() || !VFPSmallRegisterBank() then UNDEFINED;
4  lowRegsOnly = TRUE;
5  if n == 15 then UNPREDICTABLE;
```

### T2

*Armv8.1-M Main Extension only*




#### T2 variant

```
VLSTM{<c>}{<q>} <Rn>, <reglist>
```

### Decode for this encoding

```
1  if !HaveMainExt() then UNDEFINED;
2  n = UInt(Rn);
3  if !IsSecure() then UNDEFINED;
4  lowRegsOnly = FALSE;
5  if n == 15 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| <reglist> | For encoding T1: Optional register list of {D0-D15}, having the register list is the preferred disassembly<br>For encoding T2: Mandatory register list of {D0-D31} |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if CONTROL_S.SFPA == '1' then
        // Check access to the co-processor is permitted
        exc = CheckCPEnabled(10);
        HandleException(exc);

        // LSPACT should not be active at the same time as there is active FP
        // state. This is a possible attack senario so raise a SecureFault.
        lspact = if FPCCR_S.S == '1' then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
        if lspact == '1' then
            SFSR.LSERR = '1';
            exc        = CreateException(SecureFault);
            HandleException(exc);
        else
            if !IsAligned(R[n],8) then
               UFSR.UNALIGNED = '1';
               exc = CreateException(UsageFault);
               HandleException(exc);

            if FPCCR.LSPEN == '0' then
                for i = 0 to 15
                    MemA[R[n] + (4*i), 4] = S[i];
                MemA[R[n] + 0x40, 4] = FPSCR;
                if HaveMve() then
                    MemA[R[n] + 0x44, 4] = VPR;

                pushFPCalleeFrame = FPCCR.TS == '1';
                if pushFPCalleeFrame then
                    for i = 0 to 15
                        MemA[R[n] + 0x48 + (4*i), 4] = S[i+16];

                InvalidateFPRegs(pushFPCalleeFrame, pushFPCalleeFrame);

                if !lowRegsOnly && boolean UNKNOWN then
                    for i = 0 to 31
                        MemA[R[n] + 0x88 + (4*i), 4] = bits(32) UNKNOWN;
            else
                UpdateFPCCR(R[n], FALSE);

        CONTROL.FPCA = '0';
```

## C2.4.350 VMAX, VMAXA

Vector Maximum, Vector Maximum Absolute. Find the maximum value of the elements in the source operands, and store the result in the corresponding destination elements.

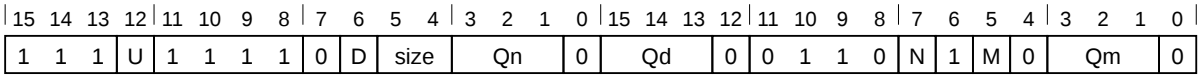
The absolute variant takes the elements from the destination vector, treating them as unsigned, and compares them to the absolute values of the corresponding elements in the source vector. The larger values are stored back into the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMAX variant

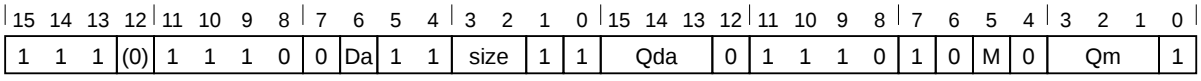
```
VMAX<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  absolute = FALSE;
8  unsigned = U == '1';
9  esize    = 8 << UInt(size);
10 elements = 32 DIV esize;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VMAXA variant

```
VMAXA<v>.<dt> Qda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || M == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(M:Qm);
6  d        = da;
7  n        = da;
8  absolute = TRUE;
9  unsigned = FALSE;
```

```
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00, U = 0 |
| U8 | Encoded as | size = 00, U = 1 |
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

## Assembler symbols for T2 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00 |
| S16 | Encoded as | size = 01 |
| S32 | Encoded as | size = 10 |

## Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Qd> Destination vector register.
<Qda> Source and destination vector register.
<Qm> Source vector register.
<Qn> Source vector register.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
for e = 0 to elements-1
    value1 = Int(Elem[op1, e, esize], unsigned || absolute);
    value2 = Int(Elem[op2, e, esize], unsigned);
    if absolute then
        value2 = Abs(value2);
    Elem[result, e, esize] = Max(value1, value2)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

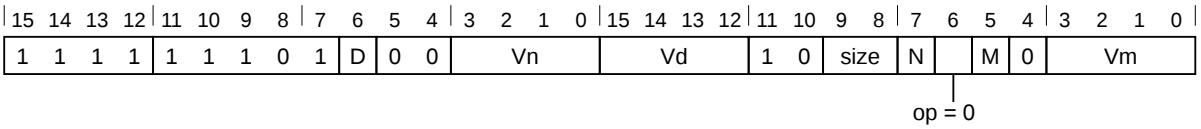
## C2.4.351 VMAXNM

Floating-point Maximum Number. Floating-point Maximum Number determines the floating-point maximum number.

NaN handling is specified by IEEE754-2008.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  maximum = (op == '0');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |

| | |
|---|---|
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();
case size of
    when '01'
        if maximum then
            S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>);
        else
            S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>);
    when '10'
        if maximum then
            S[d] = FPMaxNum(S[n], S[m]);
        else
            S[d] = FPMinNum(S[n], S[m]);
    when '11'
        if maximum then
            D[d] = FPMaxNum(D[n], D[m]);
        else
            D[d] = FPMinNum(D[n], D[m]);
```

## C2.4.352 VMAXNM, VMAXNMA (floating-point)

Vector Maximum, Vector Maximum Absolute. Find the floating-point maximum number of the elements in the source operands, and store the result in the corresponding destination elements. It handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant takes the absolute values of the elements from the destination vector and compares them to the absolute values of the corresponding elements in the source vector. The larger values are stored back into the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | 0 | D | 0 | sz | Qn | | | 0 | Qd | | | 0 | 1 | 1 | 1 | 1 | N | 1 | M | 1 | Qm | | | 0 |

#### T1: VMAXNM variant

```
VMAXNM<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d         = UInt(D:Qd);
4  m         = UInt(M:Qm);
5  n         = UInt(N:Qn);
6  absolute  = FALSE;
7  esize     = if sz == '1' then 16 else 32;
8  elements  = 32 DIV esize;
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | sz | 1 | 1 | 1 | 0 | 0 | Da | 1 | 1 | 1 | 1 | 1 | 1 | Qda | | | 0 | 1 | 1 | 1 | 0 | 1 | 0 | M | 0 | Qm | | | 1 |

#### T2: VMAXNMA variant

```
VMAXNMA<v>.<dt> Qda, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if Da == '1' || M == '1' then UNDEFINED;
3  da        = UInt(Da:Qda);
4  m         = UInt(M:Qm);
5  d         = da;
6  n         = da;
7  absolute  = TRUE;
8  esize     = if sz == '1' then 16 else 32;
9  elements  = 32 DIV esize;
```

```
10  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register. |

## Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3
4   (curBeat, elmtMask) = GetCurInstrBeat();
5
6   result = Zeros(32);
7   op1    = Q[n, curBeat];
8   op2    = Q[m, curBeat];
9   for e = 0 to elements-1
10      value1 = Elem[op1, e, esize];
11      value2 = Elem[op2, e, esize];
12      if absolute then
13          value1 = FPAbs(value1);
14          value2 = FPAbs(value2);
15      Elem[result, e, esize] = FPMaxNum(value1, value2);
16
17  for e = 0 to 3
18      if elmtMask<e> == '1' then
19          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.353 VMAXNMV, VMAXNMAV (floating-point)

Vector Maximum Across Vector, Vector Maximum Absolute Across Vector. Find the maximum value of the elements in a vector register. Store the maximum value in the general-purpose destination register only if it is larger than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. For half-precision the upper half of the general-purpose register is cleared on writeback. This instruction handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

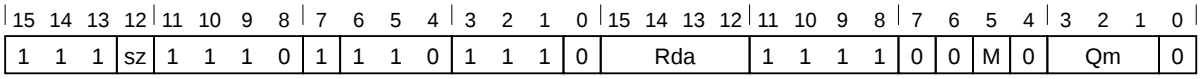
The absolute variant of the instruction compares the absolute value of vector elements.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VMAXNMV variant

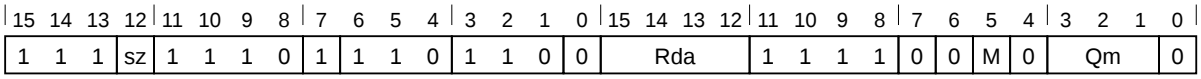
```
VMAXNMV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if M == '1' then UNDEFINED;
3  da       = UInt(Rda);
4  m        = UInt(M:Qm);
5  absolute = FALSE;
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
9  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VMAXNMAV variant

```
VMAXNMAV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if M == '1' then UNDEFINED;
3  da       = UInt(Rda);
4  m        = UInt(M:Qm);
5  absolute = TRUE;
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
9  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Rda> | General-purpose source and destination register. |
| <Qm> | Source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Elem[R[da], 0, esize];
for e = 0 to elements-1
    if elmtMask<e*(esize>>3)> == '1' then
        value = Elem[op1, e, esize];
        if absolute then
            value = FPAbs(value);
        result = FPMaxNum(value, result);

R[da] = ZeroExtend(result);
```

## C2.4.354 VMAXV, VMAXAV

Vector Maximum Across Vector, Vector Maximum Absolute Across Vector. Find the maximum value of the elements in a vector register. Store the maximum value in the general-purpose destination register only if it is larger than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. The result of the operation is sign-extended to 32 bits before being stored back.

The absolute variant of the instruction compares the absolute value of signed vector elements and treats the value in the general-purpose register as unsigned.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | size | | 1 | 0 | Rda | | | | 1 | 1 | 1 | 1 | 0 | 0 | M | 0 | Qm | | | 0 |

#### T1: VMAXV variant

```
VMAXV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  da       = UInt(Rda);
5  m        = UInt(M:Qm);
6  absolute = FALSE;
7  unsigned = U == '1';
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
11 if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | (0) | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | size | | 0 | 0 | Rda | | | | 1 | 1 | 1 | 1 | 0 | 0 | M | 0 | Qm | | | 0 |

#### T2: VMAXAV variant

```
VMAXAV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  da       = UInt(Rda);
5  m        = UInt(M:Qm);
6  absolute = TRUE;
7  unsigned = FALSE;
8  esize    = 8 << UInt(size);
```

```
elements = 32 DIV esize;
if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00, U = 0 |
| U8 | Encoded as | size = 00, U = 1 |
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

### Assembler symbols for T2 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00 |
| S16 | Encoded as | size = 01 |
| S32 | Encoded as | size = 10 |

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Rda> General-purpose source and destination register.
<Qm> Source vector register.

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Int(Elem[R[da], 0, esize], absolute || unsigned);
for e = 0 to elements-1
    if elmtMask<e*(esize>>3)> == '1' then
        value = Int(Elem[op1, e, esize], unsigned);
        if absolute then
            value = Abs(value);
        result = Max(value, result);

R[da] = result<31:0>;
```

## C2.4.355 VMIN, VMINA

Vector Minimum, Vector Minimum Absolute. Find the minimum value of the elements in the source operands, and store the result in the corresponding destination elements.

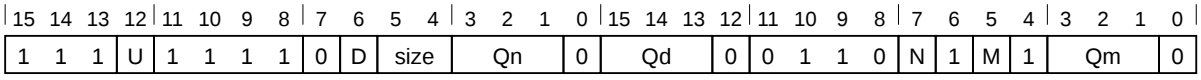
The absolute variant takes the elements from the destination vector, treating them as unsigned, and compares them to the absolute values of the corresponding elements in the source vector. The smaller values are stored back into the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMIN variant

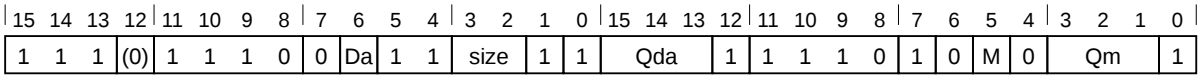
```
VMIN<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  absolute = FALSE;
8  unsigned = U == '1';
9  esize    = 8 << UInt(size);
10 elements = 32 DIV esize;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VMINA variant

```
VMINA<v>.<dt> Qda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || M == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(M:Qm);
6  d        = da;
7  n        = da;
8  absolute = TRUE;
9  unsigned = FALSE;
```

```
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

| | |
|---|---|
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |

## Assembler symbols for T2 encodings

| | |
|---|---|
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
for e = 0 to elements-1
    value1 = Int(Elem[op1, e, esize], unsigned || absolute);
    value2 = Int(Elem[op2, e, esize], unsigned);
    if absolute then
        value2 = Abs(value2);
    Elem[result, e, esize] = Min(value1, value2)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

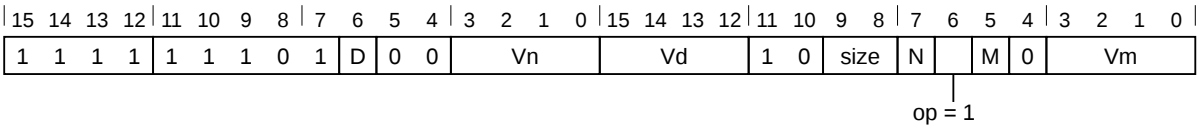
## C2.4.356 VMINNM

Floating-point Minimum Number. Floating-point Minimum Number determines the floating-point minimum number.

NaN handling is specified by IEEE754-2008.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if InITBlock() then UNPREDICTABLE;
5  maximum = (op == '0');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |

| | |
|---|---|
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();
case size of
    when '01'
        if maximum then
            S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>);
        else
            S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>);
    when '10'
        if maximum then
            S[d] = FPMaxNum(S[n], S[m]);
        else
            S[d] = FPMinNum(S[n], S[m]);
    when '11'
        if maximum then
            D[d] = FPMaxNum(D[n], D[m]);
        else
            D[d] = FPMinNum(D[n], D[m]);
```

## C2.4.357 VMINNM, VMINNMA (floating-point)

Vector Minimum, Vector Minimum Absolute. Find the floating-point minimum number of the elements in the source operands, and store the result in the corresponding destination elements. It handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

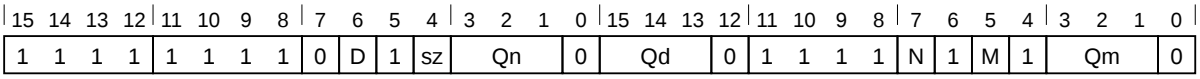
The absolute variant takes the absolute values of the elements from the destination vector and compares them to the absolute values of the corresponding elements in the source vector. The smaller values are stored back into the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VMINNM variant

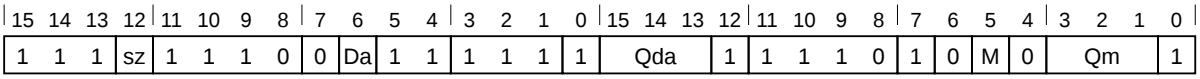
```
VMINNM<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if D == '1' || M == '1' || N == '1' then UNDEFINED;
3 d        = UInt(D:Qd);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 absolute = FALSE;
7 esize    = if sz == '1' then 16 else 32;
8 elements = 32 DIV esize;
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VMINNMA variant

```
VMINNMA<v>.<dt> Qda, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveFp);
2 if Da == '1' || M == '1' then UNDEFINED;
3 da       = UInt(Da:Qda);
4 m        = UInt(M:Qm);
5 d        = da;
6 n        = da;
7 absolute = TRUE;
8 esize    = if sz == '1' then 16 else 32;
9 elements = 32 DIV esize;
```

```
10  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register. |

## Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3
4   (curBeat, elmtMask) = GetCurInstrBeat();
5
6   result = Zeros(32);
7   op1    = Q[n, curBeat];
8   op2    = Q[m, curBeat];
9   for e = 0 to elements-1
10      value1 = Elem[op1, e, esize];
11      value2 = Elem[op2, e, esize];
12      if absolute then
13          value1 = FPAbs(value1);
14          value2 = FPAbs(value2);
15      Elem[result, e, esize] = FPMinNum(value1, value2);
16
17  for e = 0 to 3
18      if elmtMask<e> == '1' then
19          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.358 VMINNMV, VMINNMAV (floating-point)

Vector Minimum Across Vector, Vector Minimum Absolute Across Vector. Find the minimum value of the elements in a vector register. Store the minimum value in the general-purpose destination register only if it is smaller than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. For half-precision the upper half of the general-purpose register is cleared on writeback. This instruction handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant of the instruction compares the absolute value of vector elements.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | sz | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | Rda | | | | 1 | 1 | 1 | 1 | 1 | 0 | M | 0 | Qm | | | 0 |

#### T1: VMINNMV variant

```
VMINNMV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if M == '1' then UNDEFINED;
3  da       = UInt(Rda);
4  m        = UInt(M:Qm);
5  absolute = FALSE;
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
9  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | sz | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | Rda | | | | 1 | 1 | 1 | 1 | 1 | 0 | M | 0 | Qm | | | 0 |

#### T2: VMINNMAV variant

```
VMINNMAV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if M == '1' then UNDEFINED;
3  da       = UInt(Rda);
4  m        = UInt(M:Qm);
5  absolute = TRUE;
6  esize    = if sz == '1' then 16 else 32;
7  elements = 32 DIV esize;
8  if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
9  if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Rda> | General-purpose source and destination register. |
| <Qm> | Source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Elem[R[da], 0, esize];
for e = 0 to elements-1
    if elmtMask<e*(esize>>3)> == '1' then
        value = Elem[op1, e, esize];
        if absolute then
            value = FPAbs(value);
        result = FPMinNum(value, result);

R[da] = ZeroExtend(result);
```

## C2.4.359 VMINV, VMINAV

Vector Minimum Across Vector, Vector Minimum Absolute Across Vector. Find the minimum value of the elements in a vector register. Store the minimum value in the general-purpose destination register only if it is smaller than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. The result of the operation is sign-extended to 32 bits before being stored back.

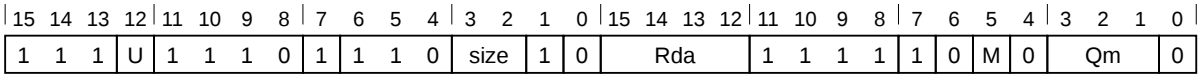
The absolute variant of the instruction compares the absolute value of signed vector elements and treats the value in the general-purpose register as unsigned.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMINV variant

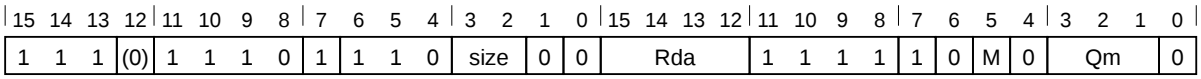
```
VMINV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  da       = UInt(Rda);
5  m        = UInt(M:Qm);
6  absolute = FALSE;
7  unsigned = U == '1';
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
11 if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VMINAV variant

```
VMINAV<v>.<dt> Rda, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if M == '1' then UNDEFINED;
4  da       = UInt(Rda);
5  m        = UInt(M:Qm);
6  absolute = TRUE;
7  unsigned = FALSE;
```

```
esize    = 8 << UInt(size);
elements = 32 DIV esize;
if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
if Rda == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

| | |
|---|---|
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |

### Assembler symbols for T2 encodings

| | |
|---|---|
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Int(Elem[R[da], 0, esize], absolute || unsigned);
for e = 0 to elements-1
    if elmtMask<e*(esize>>3)> == '1' then
        value = Int(Elem[op1, e, esize], unsigned);
        if absolute then
            value = Abs(value);
        result = Min(value, result);

R[da] = result<31:0>;
```

## C2.4.360 VMLA (vector by scalar plus vector)

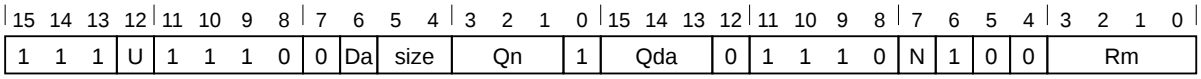
Vector Multiply Accumulate. Multiply each element in the source vector by a scalar value and add to the respective element from the destination vector. Store the result in the destination register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMLA variant

```
VMLA<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || N == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(Rm);
6  n        = UInt(N:Qn);
7  unsigned = (U == '1');
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qda> | Accumulator vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result   = Zeros(32);
```

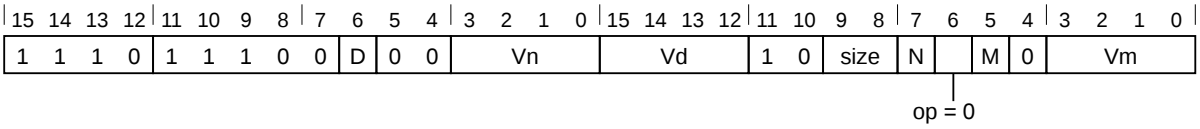
```
op1      = Q[n, curBeat];
element2 = Int(R[m]<esize-1:0>, unsigned);
op3      = Q[da, curBeat];
for e = 0 to elements-1
    element1 = Int(Elem[op1, e, esize], unsigned);
    element3 = Int(Elem[op3, e, esize], unsigned);
    value    = (element1 * element2) + element3;
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.361 VMLA

Floating-point Multiply Accumulate. Floating-point Multiply Accumulate multiplies two floating-point registers, adds the product to the destination register, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  add = (op == '0');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, TRUE) else FPNeg(FPMul(S[n
                ]<15:0>, S[m]<15:0>, TRUE));
            S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, TRUE);
        when '10'
            addend32 = if add then FPMul(S[n], S[m], TRUE) else FPNeg(FPMul(S[n], S[m], TRUE)
                );
            S[d] = FPAdd(S[d], addend32, TRUE);
        when '11'
            addend64 = if add then FPMul(D[n], D[m], TRUE) else FPNeg(FPMul(D[n], D[m], TRUE)
                );
            D[d] = FPAdd(D[d], addend64, TRUE);
```

## C2.4.362 VMLADAV

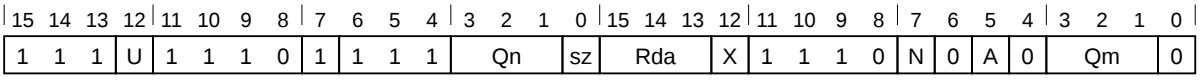
Vector Multiply Add Dual Accumulate Across Vector. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated and the lower 32 bits written back to the general-purpose destination register. The initial value of the general-purpose destination register can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMLADAV variant

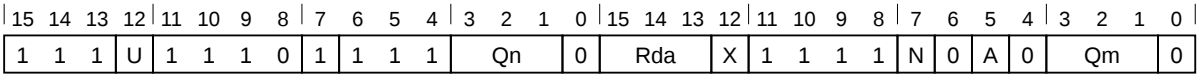
```
VMLADAV{A}{X}<v>.<dt> Rda, Qn, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if N == '1'             then UNDEFINED;
if U == '1' && X == '1' then UNDEFINED;
da          = UInt(Rda:'0');
m           = UInt(Qm);
n           = UInt(N:Qn);
exchange    = (X    == '1');
accumulate  = (A    == '1');
esize       = if sz == '0' then 16 else 32;
elements    = 32 DIV esize;
unsigned    = (U    == '1');
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VMLADAV variant

```
VMLADAV{A}{X}<v>.<dt> Rda, Qn, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if N == '1'             then UNDEFINED;
if U == '1' && X == '1' then UNDEFINED;
da          = UInt(Rda:'0');
m           = UInt(Qm);
n           = UInt(N:Qn);
exchange    = (X == '1');
```

```
accumulate = (A == '1');
esize      = 8;
elements   = 32 DIV esize;
unsigned   = (U == '1');
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S16 | Encoded as | sz = 0, U = 0 |
| U16 | Encoded as | sz = 0, U = 1 |
| S32 | Encoded as | sz = 1, U = 0 |
| U32 | Encoded as | sz = 1, U = 1 |

## Assembler symbols for T2 encodings

<dt> Unsigned flag: S indicates signed, U indicates unsigned.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | U = 0 |
| U8 | Encoded as | U = 1 |

## Assembler symbols for all encodings

<A> Accumulate with existing register contents.
This parameter must be one of the following values:

| | | |
|---|---|---|
| ” | Encoded as | A = 0 |
| A | Encoded as | A = 1 |

<X> Exchange adjacent pairs of values in Qm.
This parameter must be one of the following values:

| | | |
|---|---|---|
| ” | Encoded as | X = 0 |
| X | Encoded as | X = 1 |

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Rda> General-purpose source and destination register. This must be an even numbered register.
<Qn> First source vector register.
<Qm> Second source vector register.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then Int(R[da], unsigned) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if elmtMask<0> == '1' then
        if exchange then
            if curBeat<0> == '0' then
                mul = Int(Q[n, curBeat+1], unsigned) * Int(Q[m, curBeat], unsigned);
            else
                mul = Int(Q[n, curBeat-1], unsigned) * Int(Q[m, curBeat], unsigned);
```

```
        else
            mul = Int(Q[n, curBeat], unsigned) * Int(Q[m, curBeat], unsigned);
        result = result + mul;
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if elmtMask<e*(esize>>3)> == '1' then
            if exchange then
                if e<0> == '0' then
                    mul = (Int(Elem[op1, e+1, esize], unsigned) *
                           Int(Elem[op2, e, esize], unsigned));
                else
                    mul = (Int(Elem[op1, e-1, esize], unsigned) *
                           Int(Elem[op2, e, esize], unsigned));
            else
                mul = Int(Elem[op1, e, esize], unsigned) * Int(Elem[op2, e, esize], unsigned)
                    ;
            result = result + mul;
R[da] = result<31:0>;
```

## C2.4.363 VMLALDAV

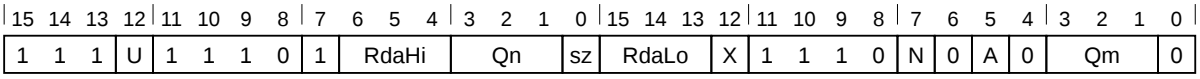
Vector Multiply Add Long Dual Accumulate Across Vector. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated. The 64 bit result is stored across two registers, the upper-half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMLALDAV variant

```
VMLALDAV{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

### Decode for this encoding

```
1  if RdaHi == '111' then SEE "VMLADAV";
2  CheckDecodeFaults(ExtType_Mve);
3  if N == '1'              then UNDEFINED;
4  if U == '1' && X == '1' then UNDEFINED;
5  dah        = UInt(RdaHi:'1');
6  dal        = UInt(RdaLo:'0');
7  m          = UInt(Qm);
8  n          = UInt(N:Qn);
9  exchange   = (X    == '1');
10 accumulate = (A    == '1');
11 esize      = if sz == '0' then 16 else 32;
12 elements   = 32 DIV esize;
13 unsigned   = (U    == '1');
14 if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
15 if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<A> Accumulate with existing register contents.
This parameter must be one of the following values:
" Encoded as A = 0
A Encoded as A = 1

<X> Exchange adjacent pairs of values in Qm.
This parameter must be one of the following values:
" Encoded as X = 0
X Encoded as X = 1

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
S16 Encoded as sz = 0, U = 0

| | |
|---|---|
| U16 | Encoded as sz = 0, U = 1 |
| S32 | Encoded as sz = 1, U = 0 |
| U32 | Encoded as sz = 1, U = 1 |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then Int(R[dah]:R[dal], unsigned) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if elmtMask<0> == '1' then
        if exchange then
            if curBeat<0> == '0' then
                mul = Int(Q[n, curBeat+1], unsigned) * Int(Q[m, curBeat], unsigned);
            else
                mul = Int(Q[n, curBeat-1], unsigned) * Int(Q[m, curBeat], unsigned);
        else
            mul = Int(Q[n, curBeat], unsigned) * Int(Q[m, curBeat], unsigned);
        result = result + mul;
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if elmtMask<e*(esize>>3)> == '1' then
            if exchange then
                if e<0> == '0' then
                    mul = (Int(Elem[op1, e+1, esize], unsigned) *
                           Int(Elem[op2, e, esize], unsigned));
                else
                    mul = (Int(Elem[op1, e-1, esize], unsigned) *
                           Int(Elem[op2, e, esize], unsigned));
            else
                mul = Int(Elem[op1, e, esize], unsigned) * Int(Elem[op2, e, esize], unsigned)
                    ;
            result = result + mul;
R[dah] = result<63:32>;
R[dal] = result<31:0>;
```

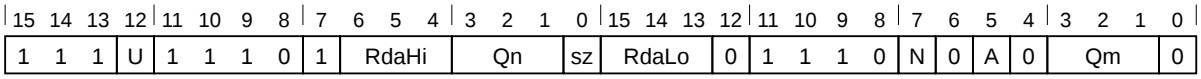
## C2.4.364 VMLALV

Vector Multiply Accumulate Long Across Vector. This is an alias of VMLALDAV without exchange.

This is an alias of VMLALDAV with the following condition satisfied: X==0.

This alias is the preferred disassembly.

### T1




#### VMLALV variant

```
VMLALV{A}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

## C2.4.365 VMLAS (vector by vector plus scalar)

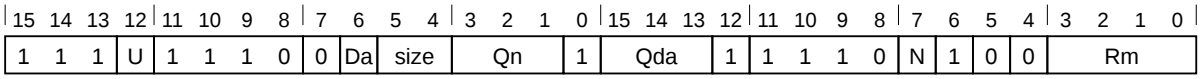
Vector Multiply Accumulate Scalar. Multiply each element in the source vector by the respective element from the destination vector and add to a scalar value. Store the result in the destination register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMLAS variant

```
VMLAS<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || N == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(Rm);
6  n        = UInt(N:Qn);
7  unsigned = (U == '1');
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qda> | Source and destination vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
5
6 result   = Zeros(32);
```

```
op1      = Q[n, curBeat];
op2      = Q[da, curBeat];
element3 = Int(R[m]<esize-1:0>, unsigned);
for e = 0 to elements-1
    element1 = Int(Elem[op1, e, esize], unsigned);
    element2 = Int(Elem[op2, e, esize], unsigned);
    value    = (element1 * element2) + element3;
    Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

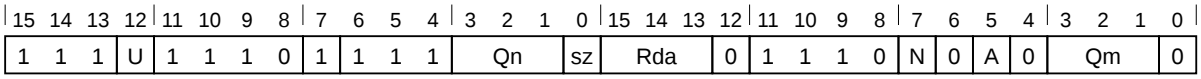
## C2.4.366 VMLAV

Vector Multiply Accumulate Across Vector. This is an alias of VMLADAV without exchange.

This is an alias of VMLADAV with the following condition satisfied: X==0.

This alias is the preferred disassembly.

### T1




#### VMLAV variant

```
VMLAV{A}<v>.<dt> Rda, Qn, Qm
```

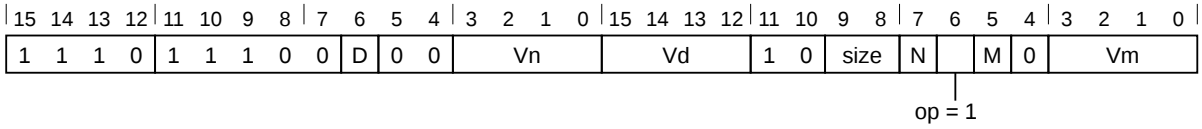
## C2.4.367 VMLS

Floating-point Multiply Subtract. Floating-point Multiply Subtract multiplies two floating-point registers, subtracts the product from the destination floating-point register, and places the result in the destination floating-point register.

Arm recommends that software does not use the VMLS instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  add = (op == '0');
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |

<Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, TRUE) else FPNeg(FPMul(S[n
                ]<15:0>, S[m]<15:0>, TRUE));
            S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, TRUE);
        when '10'
            addend32 = if add then FPMul(S[n], S[m], TRUE) else FPNeg(FPMul(S[n], S[m], TRUE)
                );
            S[d] = FPAdd(S[d], addend32, TRUE);
        when '11'
            addend64 = if add then FPMul(D[n], D[m], TRUE) else FPNeg(FPMul(D[n], D[m], TRUE)
                );
            D[d] = FPAdd(D[d], addend64, TRUE);
```

## C2.4.368 VMLSDAV

Vector Multiply Subtract Dual Accumulate Across Vector. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated and the lower 32 bits written back to the general-purpose destination register. The initial value of the general-purpose destination register can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | Qn | | | | sz | Rda | | | X | 1 | 1 | 0 | N | 0 | A | 0 | Qm | | | 1 |

#### T1: VMLSDAV variant

```
VMLSDAV{A}{X}<v>.<dt> Rda, Qn, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if N == '1' then UNDEFINED;
da          = UInt(Rda:'0');
m           = UInt(Qm);
n           = UInt(N:Qn);
exchange    = (X    == '1');
accumulate  = (A    == '1');
esize       = if sz == '0' then 16 else 32;
elements    = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | Qn | | | | 0 | Rda | | | X | 1 | 1 | 0 | N | 0 | A | 0 | Qm | | | 1 |

#### T2: VMLSDAV variant

```
VMLSDAV{A}{X}<v>.S8 Rda, Qn, Qm
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if N == '1' then UNDEFINED;
da          = UInt(Rda:'0');
m           = UInt(Qm);
n           = UInt(N:Qn);
exchange    = (X == '1');
accumulate  = (A == '1');
esize       = 8;
elements    = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

| | |
|---|---|
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S16 Encoded as sz = 0<br>S32 Encoded as sz = 1 |

## Assembler symbols for all encodings

| | |
|---|---|
| <A> | Accumulate with existing register contents.<br>This parameter must be one of the following values:<br>" Encoded as A = 0<br>A Encoded as A = 1 |
| <X> | Exchange adjacent pairs of values in Qm.<br>This parameter must be one of the following values:<br>" Encoded as X = 0<br>X Encoded as X = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rda> | General-purpose source and destination register. This must be an even numbered register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then SInt(R[da]) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if elmtMask<0> == '1' then
        if exchange then
            if curBeat<0> == '0' then
                mul = SInt(Q[n, curBeat+1]) * SInt(Q[m, curBeat]);
            else
                mul = SInt(Q[n, curBeat-1]) * SInt(Q[m, curBeat]);
        else
            mul = SInt(Q[n, curBeat]) * SInt(Q[m, curBeat]);
        if curBeat<0> == '0' then
            result = result + mul;
        else
            result = result - mul;
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if elmtMask<e*(esize>>3)> == '1' then
            if exchange then
                if e<0> == '0' then
                    mul = (SInt(Elem[op1, e+1, esize]) *
                           SInt(Elem[op2, e, esize]));
                else
                    mul = (SInt(Elem[op1, e-1, esize]) *
                           SInt(Elem[op2, e, esize]));
            else
                mul = SInt(Elem[op1, e, esize]) * SInt(Elem[op2, e, esize]);
            if e<0> == '0' then
                result = result + mul;
```

```
            else
                result = result - mul;
R[da] = result<31:0>;
```

## C2.4.369 VMLSLDAV

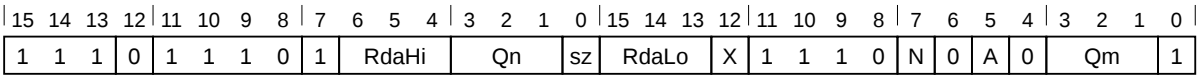
Vector Multiply Subtract Long Dual Accumulate Across Vector. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated. The 64 bit result is stored across two registers, the upper-half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMLSLDAV variant

```
VMLSLDAV{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

### Decode for this encoding

```
1  if RdaHi == '111' then SEE "VMLSDAV";
2  CheckDecodeFaults(ExtType_Mve);
3  if N == '1' then UNDEFINED;
4  dah        = UInt(RdaHi:'1');
5  dal        = UInt(RdaLo:'0');
6  m          = UInt(Qm);
7  n          = UInt(N:Qn);
8  exchange   = (X    == '1');
9  accumulate = (A    == '1');
10 esize      = if sz == '0' then 16 else 32;
11 elements   = 32 DIV esize;
12 if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
13 if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <A> | Accumulate with existing register contents.<br>This parameter must be one of the following values:<br>" Encoded as A = 0<br>A Encoded as A = 1 |
| <X> | Exchange adjacent pairs of values in Qm.<br>This parameter must be one of the following values:<br>" Encoded as X = 0<br>X Encoded as X = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S16 Encoded as sz = 0<br>S32 Encoded as sz = 1 |
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register. |

| | |
|---|---|
| <RdaHi> | General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

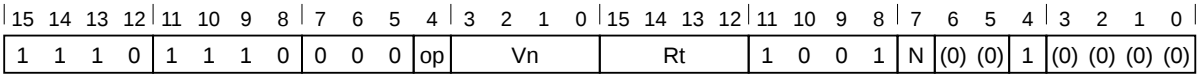
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then SInt(R[dah]:R[dal]) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if elmtMask<0> == '1' then
        if exchange then
            if curBeat<0> == '0' then
                mul = SInt(Q[n, curBeat+1]) * SInt(Q[m, curBeat]);
            else
                mul = SInt(Q[n, curBeat-1]) * SInt(Q[m, curBeat]);
        else
            mul = SInt(Q[n, curBeat]) * SInt(Q[m, curBeat]);
        if curBeat<0> == '0' then
            result = result + mul;
        else
            result = result - mul;
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if elmtMask<e*(esize>>3)> == '1' then
            if exchange then
                if e<0> == '0' then
                    mul = (SInt(Elem[op1, e+1, esize]) *
                           SInt(Elem[op2, e, esize]));
                else
                    mul = (SInt(Elem[op1, e-1, esize]) *
                           SInt(Elem[op2, e, esize]));
            else
                mul = SInt(Elem[op1, e, esize]) * SInt(Elem[op2, e, esize]);
            if e<0> == '0' then
                result = result + mul;
            else
                result = result - mul;
R[dah] = result<63:32>;
R[dal] = result<31:0>;
```

## C2.4.370 VMOV (between general-purpose register and half-precision register)

Floating-point Move (between general-purpose register and half-precision register). Floating-point Move (between general-purpose register and half-precision register) transfers the contents of a half-precision register to a general-purpose register, or the contents of a general-purpose register to a half-precision register

### T1

*Armv8.1-M Floating-point Extension*




**T1 variant**

**From general-purpose register `op == 0`.**

```
VMOV{<c>}{<q>}.F16, <Sn>, <Rt>
```

**T1 variant**

**To general-purpose register `op == 1`.**

```
VMOV{<c>}{<q>}.F16, <Rt>, <Sn>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_HpFp);
2 if InITBlock() then UNPREDICTABLE;
3 to_arm_register = (op == '1');
4 t = UInt(Rt); n = UInt(Vn:N);
5 if t == 15 || t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sn> | Is the 32-bit name of the floating-point source register, encoded in the "Vn:N" field. |
| <Rt> | Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field. |

### Operation for all encodings

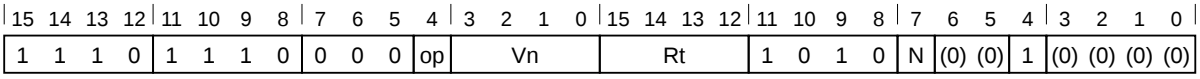
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteFPCheck();
4     if to_arm_register then
5         R[t] = Zeros(16) : S[n]<15:0>;
6     else
7         S[n] = Zeros(16) : R[t]<15:0>;
```

## C2.4.371 VMOV (between general-purpose register and single-precision register)

Floating-point Move (between general-purpose register and single-precision register). Floating-point Move (between general-purpose register and single-precision register) transfers the contents of a single-precision register to a general-purpose register, or the contents of a general-purpose register to a single-precision register.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Encoding**

**Applies when `op == 0`.**

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

**Encoding**

**Applies when `op == 1`.**

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveOrFp);
2 to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
3 if t == 15 || t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <Rt> | Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field. |
| <Sn> | Is the 32-bit name of the floating-point register to be transferred, encoded in the "Vn:N" field. |
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

### Operation for all encodings

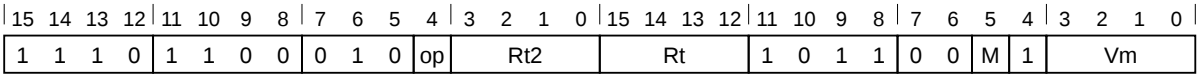
```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteFPCheck();
4     if to_arm_register then
5         R[t] = S[n];
6     else
7         S[n] = R[t];
```

## C2.4.372 VMOV (between two general-purpose registers and a doubleword register)

Floating-point Move (between two general-purpose registers and a doubleword register). Floating-point Move (between two general-purpose registers and a doubleword register) transfers two words from two general-purpose registers to a doubleword register, or from a doubleword register to two general-purpose registers.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Encoding**

**Applies when `op == 1`.**

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

**Encoding**

**Applies when `op == 0`.**

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveOrFp);
2  if VFPSmallRegisterBank() && (M == '1') then UNDEFINED;
3  to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
4  if t == 15 || t2 == 15 then UNPREDICTABLE;
5  if t == 13 || t2 == 13 then UNPREDICTABLE;
6  if to_arm_registers && t == t2 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

### Assembler symbols for all encodings

| | |
|---|---|
| <Dm> | Is the 64-bit name of the floating-point register to be transferred, encoded in the "M:Vm" field. |
| <Rt2> | Is the first general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt" field. |
| <Rt> | Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field. |
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4      if to_arm_registers then
5          R[t] = D[m]<31:0>;
```

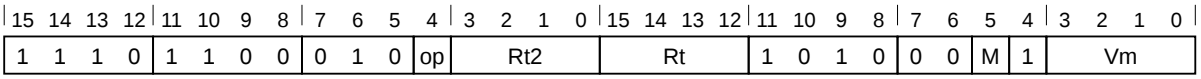
```
6           R[t2] = D[m]<63:32>;
7       else
8           D[m]<31:0> = R[t];
9           D[m]<63:32> = R[t2];
```

## C2.4.373 VMOV (between two general-purpose registers and two single-precision registers)

Floating-point Move (between two general-purpose registers and two single-precision registers). Floating-point Move (between two general-purpose registers and two single-precision registers) transfers the contents of two consecutively numbered single-precision registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision registers. The general-purpose registers do not have to be contiguous.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Encoding**

**Applies when `op == 1`.**

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

**Encoding**

**Applies when `op == 0`.**

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_MveOrFp);
2 to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
3 if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
4 if t == 13 || t2 == 13 then UNPREDICTABLE;
5 if to_arm_registers && t == t2 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `to_arm_registers && t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

#### CONSTRAINED UNPREDICTABLE behavior

If `m == 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- One or more of the single-precision registers become UNKNOWN for a move to the single-precision register. The general-purpose registers listed in the instruction become UNKNOWN for a move from the single-precision registers. This behavior does not affect any other general-purpose registers.

### Assembler symbols for all encodings

| | |
|---|---|
| `<Rt2>` | Is the second general-purpose register that `<Sm1>` will be transferred to or from, encoded in the "Rt" field. |
| `<Rt>` | Is the first general-purpose register that `<Sm>` will be transferred to or from, encoded in the "Rt" field. |

<Sm1> Is the 32-bit name of the second floating-point register to be transferred. This is the next floating-point register after <Sm>.

<Sm> Is the 32-bit name of the first floating-point register to be transferred, encoded in the "Vm:M" field.

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.

<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_arm_registers then
        R[t] = S[m];
        R[t2] = S[m+1];
    else
        S[m] = R[t];
        S[m+1] = R[t2];
```

## C2.4.374 VMOV (general-purpose register to vector lane)

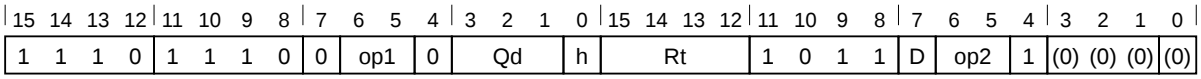
Vector Move (general-purpose register to vector lane). Copy the value of a general-purpose register to a vector lane.

This instruction is subject to beat-wise execution if it is not in an IT block.

This instruction is not VPT compatible.

### T1

*Armv8.1-M Floating-point Extension and / or Armv8.1-M MVE*




#### T1: VMOV variant

```
VMOV<c>.<dt> Qd[idx], Rt
```

### Decode for this encoding

```
1  if D   == '1'                then UNDEFINED;
2  if op1 == '0x' && op2 == '10' then UNDEFINED;
3  d = UInt(D:Qd);
4  t = UInt(Rt);
5  case (h:op1:op2) of
6      when 'x1xxx' isMve = TRUE;  esize = 8;  index = UInt((h:op1:op2)<1:0>);
7      when 'x0xx1' isMve = TRUE;  esize = 16; index = UInt((h:op1:op2)<1>);
8      when 'x0x00' isMve = FALSE; esize = 32; index = 0;
9  CheckDecodeFaults(if isMve then ExtType_Mve else ExtType_MveOrFp);
10 targetBeat = UInt((h:op1:op2)<4>:(h:op1:op2)<2>);
11 if Rt == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>32 Encoded as op1 = 0x, op2 = 00<br>16 Encoded as op1 = 0x, op2 = x1<br>8 Encoded as op1 = 1x, op2 = xx |
| <Qd> | Destination vector register. |
| <idx> | Element index to select in the vector register, must be in the range 0 to ((128/dt)-1). This value is encoded into the bits of h:op1:op2 which are not used to encode dt. |
| <Rt> | Source general-purpose register. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4
5      if InITBlock() || !HaveMve() then
6          Elem[Q[d, targetBeat],index,esize] = R[t]<esize-1:0>;
7      else
8          (curBeat, -) = GetCurInstrBeat();
9          if curBeat == targetBeat then
10             Elem[Q[d, curBeat],index,esize] = R[t]<esize-1:0>;
```

### C2.4.375 VMOV (half of doubleword register to single general-purpose register)

Floating-point Move (half of doubleword register to single general-purpose register). Floating-point Move (half of doubleword register to single general-purpose register) transfers one word from the upper or lower half of a doubleword register to a general-purpose register.

This instruction is an alias of VMOV (vector lane to general-purpose register)

## C2.4.376 VMOV (immediate) (vector)

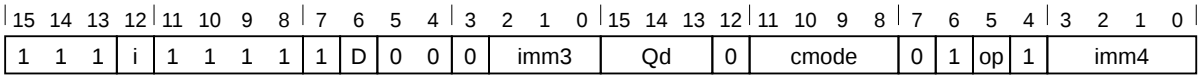
Vector Move (immediate). Set each element of a vector register to the immediate operand value. The immediate is generated by the AdvSIMDExpandImm() function based on the requested data type and immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMOV variant

```
VMOV<v>.<dt> Qd, #<imm>
```

### Decode for this encoding

```
1  if op == '0' && cmode IN {'0xx1', 'x0x1'}                 then SEE "VORR (immediate)";
2  if op == '1' && cmode IN {'0xx0', 'x0x0', 'xx00', '1101'} then SEE "VMVN (immediate)";
3  if op == '1' && cmode IN {'0xx1', 'x0x1', 'xx01'}         then SEE "VBIC (immediate)";
4  CheckDecodeFaults(ExtType_Mve);
5  if D     == '1'                 then UNDEFINED;
6  if cmode == '11x1' && op == '1' then UNDEFINED;
7  d     = UInt(D:Qd);
8  imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector, for use with the AdvSIMDExpandImm() function.
This parameter must be one of the following values:

- I32 Encoded as:
  - cmode = 0000, op = 0
  - cmode = 0010, op = 0
  - cmode = 0100, op = 0
  - cmode = 0110, op = 0
  - cmode = 1100, op = 0
  - cmode = 1101, op = 0
- I16 Encoded as:
  - cmode = 1000, op = 0
  - cmode = 1010, op = 0
- I8 Encoded as:
  - cmode = 1110, op = 0
- I64 Encoded as:
  - cmode = 1110, op = 1
- F32 Encoded as:
  - cmode = 1111, op = 0

<Qd> Destination vector register.

<imm> The immediate value to load in to each element. This must be an immediate that can be encoded for use with the AdvSIMDExpandImm() function.

## Operation for all encodings

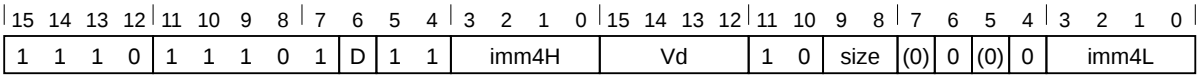
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

if curBeat<0> == '0' then
    result = imm64<31:0>;
else
    result = imm64<63:32>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.377 VMOV (immediate)

Floating-point Move (immediate). Floating-point Move (immediate) places an immediate constant into the destination floating-point register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMOV{<c>}{<q>}.F16 <Sd>, #<imm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
case size of
    when '01'
        d = UInt(Vd:D);
        imm16 = VFPExpandImm(imm4H:imm4L, 16);
        imm32 = Zeros(16) : imm16;
    when '10'
        d = UInt(Vd:D);
        imm32 = VFPExpandImm(imm4H:imm4L, 32);
    when '11'
        d = UInt(D:Vd);
        imm64 = VFPExpandImm(imm4H:imm4L, 64);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <imm> | Is a floating-point constant. For details of the range of constants available and the encoding of <imm>, see the definition of VFPExpandImm(). |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = imm64;
    else
        S[d] = imm32;
```

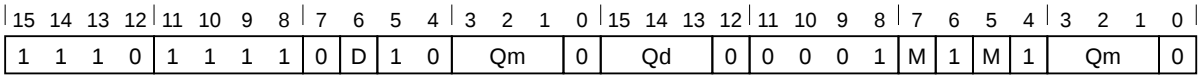
## C2.4.378 VMOV (register) (vector)

Vector Move (register). Copy the value of one vector register to another vector register.

This is an alias of VORR with the following condition satisfied: Qm==Qn.

This alias is the preferred disassembly.

### T1




**VMOV variant**

`VMOV<v> Qd, Qm`

**is equivalent to**

`VORR<v> Qd, Qm, Qm`

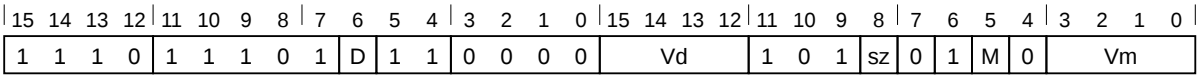
**and is the preferred disassembly when Qm == Qn**

## C2.4.379 VMOV (register)

Floating-point Move (register). Floating-point Move (register) copies the contents of one register to another.

### T2

*Armv8-M Floating-point Extension or MVE*, D == 1 UNDEFINED in single-precision only implementations.




**Single-precision scalar variant**

***Armv8-M Floating-point Extension or MVE.***

**Applies when `sz == 0`.**

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension or MVE.***

**Applies when `sz == 1`.**

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (sz == '1');
2 CheckDecodeFaults(if dp_operation then ExtType_MveOrDpFp else ExtType_MveOrFp);
3 if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
5 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteFPCheck();
4     if dp_operation then
5         D[d] = D[m];
6     else
7         S[d] = S[m];
```

### C2.4.380 VMOV (single general-purpose register to half of doubleword register)

Floating-point Move (single general-purpose register to half of doubleword register). Floating-point Move (single general-purpose register to half of doubleword register) transfers one word from a general-purpose register to the upper or lower half of a doubleword register.

This instruction is an alias of VMOV (general-purpose register to vector lane)

## C2.4.381 VMOV (two 32 bit vector lanes to two general-purpose registers)

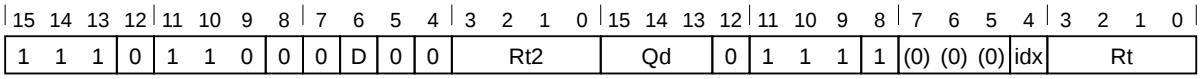
Vector Move (two 32 bit vector lanes to two general-purpose registers). Copy two 32 bit vector lanes to two general-purpose registers.

This instruction is subject to beat-wise execution if it is not in an IT block.

This instruction is not VPT compatible.

### T1

*Armv8.1-M MVE*




#### T1: VMOV variant

```
VMOV<c> Rt, Rt2, Qd[idx], Qd[idx2]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' then UNDEFINED;
3  d  = UInt(D:Qd);
4  t  = UInt(Rt);
5  t2 = UInt(Rt2);
6  if Rt  == '11x1'              then CONSTRAINED_UNPREDICTABLE;
7  if Rt2 == '11x1' || Rt == Rt2 then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Destination general-purpose register |
| <Rt2> | Destination general-purpose register |
| <idx> | The first index for the vector register.<br>This parameter must be one of the following values:<br>2 Encoded as idx = 0<br>3 Encoded as idx = 1 |
| <Qd> | Source vector register. |
| <idx2> | The second index for the vector register. This must be two less than the first index.<br>This parameter must be one of the following values:<br>0 Encoded as idx = 0<br>1 Encoded as idx = 1 |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4
5      if InITBlock() then
6          R[t]  = Q[d, UInt('0':idx)];
7          R[t2] = Q[d, UInt('1':idx)];
8      else
9          (curBeat, -) = GetCurInstrBeat();
10         if curBeat<0> == idx then
11             tReg    = if curBeat<1> == '0' then t else t2;
12             R[tReg] = Q[d, curBeat];
```

## C2.4.382 VMOV (two general-purpose registers to two 32 bit vector lanes)

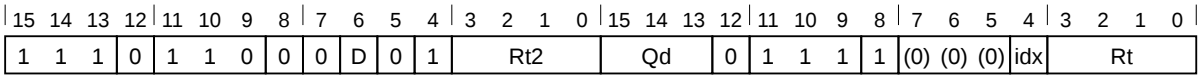
Vector Move (two general-purpose registers to two 32 bit vector lanes). Copy two general-purpose registers to two 32 bit vector lanes.

This instruction is subject to beat-wise execution if it is not in an IT block.

This instruction is not VPT compatible.

### T1

*Armv8.1-M MVE*




#### T1: VMOV variant

```
VMOV<c> Qd[idx], Qd[idx2], Rt, Rt2
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' then UNDEFINED;
3  d  = UInt(D:Qd);
4  t  = UInt(Rt);
5  t2 = UInt(Rt2);
6  if Rt  == '11x1' then CONSTRAINED_UNPREDICTABLE;
7  if Rt2 == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qd> | Destination vector register. |
| <idx> | The first index for the vector register.<br>This parameter must be one of the following values:<br>2 Encoded as idx = 0<br>3 Encoded as idx = 1 |
| <idx2> | The second index for the vector register. This must be two less than the first index.<br>This parameter must be one of the following values:<br>0 Encoded as idx = 0<br>1 Encoded as idx = 1 |
| <Rt> | Source general-purpose register. |
| <Rt2> | Source general-purpose register. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4
5      if InITBlock() then
6          Q[d, UInt('0':idx)] = R[t];
7          Q[d, UInt('1':idx)] = R[t2];
8      else
9          (curBeat, -) = GetCurInstrBeat();
10         if curBeat<0> == idx then
11             tReg          = if curBeat<1> == '0' then t else t2;
12             Q[d, curBeat] = R[tReg];
```

## C2.4.383 VMOV (vector lane to general-purpose register)

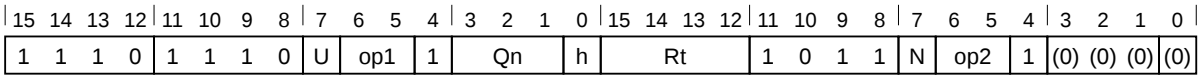
Vector Move (vector lane to general-purpose register). Copy the value of a vector lane to a general-purpose register.

This instruction is subject to beat-wise execution if it is not in an IT block.

This instruction is not VPT compatible.

### T1

*Armv8.1-M Floating-point Extension and / or Armv8.1-M MVE*




#### T1: VMOV variant

```
VMOV<c>.<dt> Rt, Qn[idx]
```

### Decode for this encoding

```
1  if N   == '1'                                then UNDEFINED;
2  if U   == '1' && op1 == '0x' && op2 == '00' then UNDEFINED;
3  if op1 == '0x' && op2 == '10'                then UNDEFINED;
4  n = UInt(N:Qn);
5  t = UInt(Rt);
6  case (U:h:op1:op2) of
7      when 'xx1xxx' isMve = TRUE;  esize = 8;  index = UInt((h:op1:op2)<1:0>);
8      when 'xx0xx1' isMve = TRUE;  esize = 16; index = UInt((h:op1:op2)<1>);
9      when '0x0x00' isMve = FALSE; esize = 32; index = 0;
10 CheckDecodeFaults(if isMve then ExtType_Mve else ExtType_MveOrFp);
11 targetBeat = UInt((h:op1:op2)<4>:(h:op1:op2)<2>);
12 unsigned   = (U == '1');
13 if Rt == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>32 Encoded as op1 = 0x, op2 = 00, U = 0<br>S16 Encoded as op1 = 0x, op2 = x1, U = 0<br>U16 Encoded as op1 = 0x, op2 = x1, U = 1<br>S8 Encoded as op1 = 1x, op2 = xx, U = 0<br>U8 Encoded as op1 = 1x, op2 = xx, U = 1 |
| <Rt> | Destination general-purpose register. |
| <Qn> | Source vector register. |
| <idx> | Element index to select in the vector register, must be in the range 0 to ((128/dt)-1). This value is encoded into the bits of h:op1:op2 which are not used to encode dt. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4
5      if InITBlock() || !HaveMve() then
```

```
6           R[t] = Extend(Elem[Q[n, targetBeat],index,esize], unsigned);
7       else
8           (curBeat, -) = GetCurInstrBeat();
9           if curBeat == targetBeat then
10              R[t] = Extend(Elem[Q[n, curBeat],index,esize], unsigned);
```

## C2.4.384 VMOVL

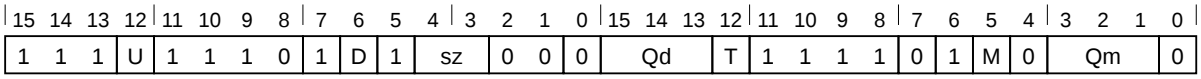
Vector Move Long. Selects an element of 8 or 16-bits from either the top half (T variant) or bottom half (B variant) of each source element, sign or zero-extends, performs a signed or unsigned left shift by an immediate value and places the 16 or 32-bit results in the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMOVL variant

```
VMOVL<T><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  if sz IN {'11', '00'} then SEE "VSHLL";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  unsigned = (U == '1');
7  esize    = 8 * UInt(sz);
8  elements = 16 DIV esize;
9  top      = UInt(T);
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the source element is used.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– The size of the elements in the vector.
This parameter must be one of the following values:
- S8 Encoded as sz = 01, U = 0
- U8 Encoded as sz = 01, U = 1
- S16 Encoded as sz = 10, U = 0
- U16 Encoded as sz = 10, U = 1

<Qd> Destination vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
```

```
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      operand = Int(Elem[op1, 2*e + top, esize], unsigned);
10     Elem[result, e, 2*esize] = operand<(2*esize)-1:0>;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.385 VMOVN

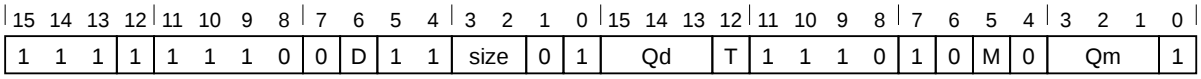
Vector Move and Narrow. Performs an element-wise narrowing to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMOVN variant

```
VMOVN<T><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D    == '1' || M == '1' then UNDEFINED;
4  if size == '10'            then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(M:Qm);
7  esize    = 8 << UInt(size);
8  elements = 16 DIV esize;
9  top      = UInt(T);
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- I16 Encoded as size = 00
- I32 Encoded as size = 01

<Qd> Destination vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  op1    = Q[m, curBeat];
7  result = Q[d, curBeat];
```

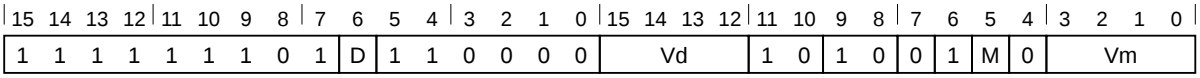
```
8  for e = 0 to elements-1
9      operand = UInt(Elem[op1, e, 2*esize]);
10     Elem[result, 2*e + top, esize] = operand<esize-1:0>;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.386 VMOVX

Floating-point Move extraction. Floating-point Move extraction copies the upper 16 bits of the 32-bit source FP register into the lower 16 bits of the 32-bit destination FP register, while clearing the remaining bits to zero.

### T1

*Armv8.1-M Floating-point Extension*




#### T1 variant

`VMOVX<q>.F16 <Sd>, <Sm>`

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_HpFp);
2 if InITBlock() then UNPREDICTABLE;
3 d = UInt(Vd:D); m = UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
2     EncodingSpecificOperations();
3     ExecuteFPCheck();
4     S[d] = Zeros(16) : S[m]<31:16>;
```

## C2.4.387 VMRS

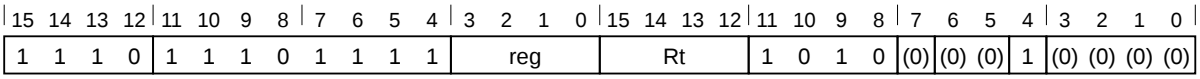
Move to general-purpose Register from Floating-point Special register. Move to general-purpose Register from Floating-point Special register moves the value of FPSCR, FPCXT, VPR, or VPR.P0 to a general-purpose register, or thevalues of FPSCR condition flags to the APSR condition flags.

If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0. This instruction is UNDEFINED if executed in Non-secure state.

### T1

*Armv8-M Floating-point Extension and / or Armv8.1-M MVE.*

- For Armv8.1-M, the `reg` field is configurable, and all of the listed registers can be accessed.
- For Armv8.0-M, only access to FPSCR is permitted, and the `reg` field is not configurable.




#### T1 variant

`VMRS{<c>}{<q>} <Rt>, <spec_reg>`

### Decode for this encoding

```
fpCxtAccess    = HasArchVersion(Armv8p1) && (reg == '111x');
fpCxtNSSAccess = HasArchVersion(Armv8p1) && (reg == '1110');
fpInactive     = !HaveMveOrFPExt()       || (FPCCR_NS.ASPEN == '1' && CONTROL.FPCA == '0');
if fpCxtAccess then
    if !HaveMainExt() || !IsSecure() then UNDEFINED;
    if !fpInactive then
        HandleException(CheckCPEnabled(10));
else
    CheckDecodeFaults(ExtType_MveOrFp);
t = UInt(Rt);
if t == 13 || (t == 15 && reg != '0001') then UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Rt> | Is the general-purpose destination register, encoded in the "Rt" field. Is one of:<br>APSR_nzcv Encoded as 0b1111. This instruction transfers the FPSCR.N, Z, C, V condition flags to the APSR.N, Z, C, V condition flags.<br>R0-R14 General-purpose register. |
| <spec_reg> | Is the special register to be accessed, encoded in the "reg" field. The permitted values are:<br>0b1110 FPCXT_NS, enables saving and resoration of the Non-secure floating-point context.<br>0b1111 FPCXT_S, enables saving and restoration of the Secure floating-point context.<br>0b0001 FPSCR.<br>0b0010 FPSCR_nzcvqc, access to FPSCR condition and saturation flags.<br>0b1101 P0, access to VPR.P0 predicate field.<br>0b1100 VPR, privileged only acccess to the VPR register. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !fpCxtNSSAccess then
```

```
        ExecuteFPCheck();
    elsif !fpInactive then
        PreserveFPState();
    SerializeVFP();
    VFPExcBarrier();

    case reg of
        when '0001'
            if t == 15 then
                APSR.N = FPSCR.N;
                APSR.Z = FPSCR.Z;
                APSR.C = FPSCR.C;
                APSR.V = FPSCR.V;
            else
                R[t] = FPSCR;
        when '0010'
            if HasArchVersion(Armv8p1) then
                // Only read the N, Z, C, V, and QC flags
                R[t] = FPSCR<31:27>:Zeros(27);
            else
                UNPREDICTABLE;
        when '1100'
            if HaveMve() then
                if CurrentModeIsPrivileged() then
                    R[t] = VPR;
            else
                UNPREDICTABLE;
        when '1101'
            if HaveMve() then
                R[t] = Zeros(16):VPR.P0;
            else
                UNPREDICTABLE;
        when '1110'
            if HasArchVersion(Armv8p1) then
                if HaveFPExt() || HaveMve() then
                    FPCXT_Type cxt = Zeros(32);
                    if !fpInactive then
                        cxt.SFPA  = CONTROL_S.SFPA;
                        cxt<27:0> = FPSCR<27:0>;
                        // If the FP context isn't secure the FPSCR value is set
                        // to the NS default so any NS functions that are called
                        // before an FP instruction is executed in the secure
                        // state will get the same FPSCR value as functions
                        // called after a secure FP instruction (I.E. the value
                        // of FPDSCR_NS).
                        if CONTROL_S.SFPA == '0' then
                            FPSCR = FPDSCR_NS<31:0>;
                    else
                        cxt<27:0> = FPDSCR_NS<27:0>;
                    R[t] = cxt;
            else
                UNPREDICTABLE;
        when '1111'
            if HasArchVersion(Armv8p1) then
                FPCXT_Type cxt = Zeros(32);
                cxt.SFPA       = CONTROL_S.SFPA;
                cxt<27:0>      = FPSCR<27:0>;
                FPSCR          = FPDSCR_NS<31:0>;
                CONTROL_S.SFPA = '0';
                R[t]           = cxt;
            else
                UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
```

## C2.4.388 VMSR

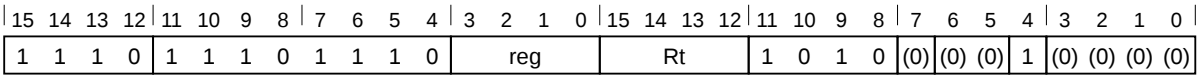
Move to Floating-point Special register from general-purpose Register. Move to Floating-point Special register from general-purpose Register moves the value of a general-purpose register to FPSCR, FPCXT, VPR, or VPR.P0.

If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0. This instruction is UNDEFINED if executed in Non-secure state.

### T1

*Armv8-M Floating-point Extension and / or Armv8.1-M MVE.*

- For Armv8.1-M, the `reg` field is configurable, and all of the listed registers can be accessed.
- For Armv8.0-M, only access to FPSCR is permitted, and the `reg` field is not configurable.




#### T1 variant

`VMSR{<c>}{<q>} <spec_reg>, <Rt>`

### Decode for this encoding

```
fpCxtAccess    = HasArchVersion(Armv8p1) && (reg == '111x');
fpCxtNSSAccess = HasArchVersion(Armv8p1) && (reg == '1110');
fpInactive     = !HaveMveOrFPExt()       || (FPCCR_NS.ASPEN == '1' && CONTROL.FPCA == '0');
if fpCxtAccess then
    if !HaveMainExt() || !IsSecure() then UNDEFINED;
    if !fpInactive then
        HandleException(CheckCPEnabled(10));
else
    CheckDecodeFaults(ExtType_MveOrFp);
t = UInt(Rt);
if t == 15 || t == 13 then UNPREDICTABLE;
```

### Assembler symbols for all encodings

`<c>` See C1.2.5 *Standard assembler syntax fields* on page 424.

`<q>` See C1.2.5 *Standard assembler syntax fields* on page 424.

`<spec_reg>` Is the special register to be accessed, encoded in the "reg" field. The permitted values are:

- `0b1110` FPCXT_NS, enables saving and resoration of the Non-secure floating-point context.
- `0b1111` FPCXT_S, enables saving and restoration of the Secure floating-point context.
- `0b0001` FPSCR.
- `0b0010` FPSCR_nzcvqc, access to FPSCR condition and saturation flags.
- `0b1101` P0, access to VPR.P0 predicate field.
- `0b1100` VPR, privileged only acccess to the VPR register.

`<Rt>` Is the general-purpose source register to be transferred to `<spec_reg>`, encoded in the "Rt" field.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !fpCxtNSSAccess then
        ExecuteFPCheck();
    elsif !fpInactive then
        PreserveFPState();
    SerializeVFP();
```

```
    VFPExcBarrier();

    case reg of
        when '0001'
            FPSCR = R[t];
        when '0010'
            if HasArchVersion(Armv8p1) then
                // Only update the N, Z, C, V, and QC flags
                FPSCR<31:27> = R[t]<31:27>;
            else
                UNPREDICTABLE;
        when '1100'
            if HaveMve() then
                if CurrentModeIsPrivileged() then
                    VPR = R[t];
            else
                UNPREDICTABLE;
        when '1101'
            if HaveMve() then
                VPR.P0 = R[t]<15:0>;
            else
                UNPREDICTABLE;
        when '1110'
            if HasArchVersion(Armv8p1) then
                if (HaveFPExt() || HaveMve()) && !fpInactive then
                    FPCXT_Type cxt = R[t];
                    CONTROL_S.SFPA = cxt.SFPA;
                    FPSCR          = Zeros(4):cxt<27:0>;
            else
                UNPREDICTABLE;
        when '1111'
            if HasArchVersion(Armv8p1) then
                FPCXT_Type cxt = R[t];
                CONTROL_S.SFPA = cxt.SFPA;
                FPSCR          = Zeros(4):cxt<27:0>;
            else
                UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;
```

## C2.4.389 VMUL (floating-point)

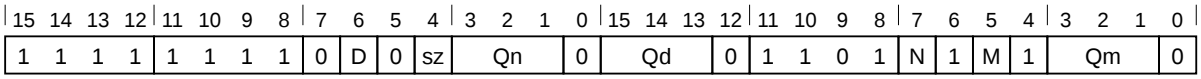
Vector Multiply. Multiply the value of the elements in the first source vector register by either the respective elements in the second source vector register or a general-purpose register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VMUL variant

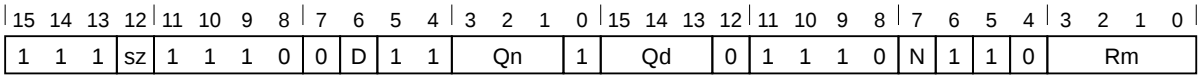
```
VMUL<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(M:Qm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = FALSE;
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VMUL variant

```
VMUL<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(Rm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = TRUE;
9  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
10 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = FPMul(Elem[op1, e, esize], R[m]<esize-1:0>, FALSE);
        Elem[result, e, esize] = value;
else
    for e = 0 to elements-1
        op2   = Q[m, curBeat];
        value = FPMul(Elem[op1, e, esize], Elem[op2, e, esize], FALSE);
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.390 VMUL (vector)

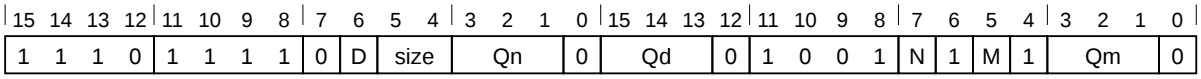
Vector Multiply. Multiply the value of the elements in the first source vector register by either the respective elements in the second source vector register or a general-purpose register. The result is then written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMUL variant

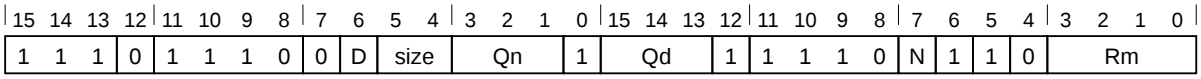
```
VMUL<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VMUL variant

```
VMUL<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  withScalar = TRUE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>I8 Encoded as size = 00<br>I16 Encoded as size = 01<br>I32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

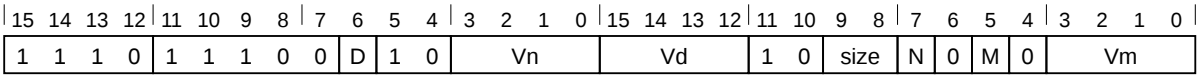
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = SInt(Elem[op1, e, esize]) * SInt(R[m]<esize-1:0>);
        Elem[result, e, esize] = value<esize-1:0>;
else
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        value = SInt(Elem[op1, e, esize]) * SInt(Elem[op2, e, esize]);
        Elem[result, e, esize] = value<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.391 VMUL

Floating-point Multiply. Floating-point Multiply multiplies two floating-point register values, and places the result in the destination floating-point register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VMUL{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

### Decode for this encoding

```
1 dp_operation = (size == '11');
2 CheckFPDecodeFaults(size);
3 if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
      UNDEFINED;
4 if size == '01' && InITBlock() then UNPREDICTABLE;
5 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6 n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1 if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            S[d] = Zeros(16) : FPMul(S[n]<15:0>, S[m]<15:0>, TRUE);
        when '10'
            S[d] = FPMul(S[n], S[m], TRUE);
        when '11'
            D[d] = FPMul(D[n], D[m], TRUE);
```

## C2.4.392 VMULH, VRMULH

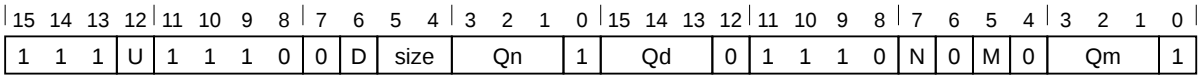
Vector Multiply Returning High Half, Vector Rounding Multiply Returning High Half. Multiply each element in a vector register by its respective element in another vector register and return the high half of the result. The result is optionally rounded before the high half is selected.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMULH variant

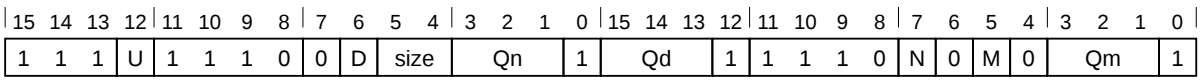
```
VMULH<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  unsigned = (U == '1');
10 round    = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VRMULH variant

```
VRMULH<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  unsigned = (U == '1');
10 round    = TRUE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
rVal   = if round then 1 << (esize-1) else 0;
for e = 0 to elements-1
    value = (Int(Elem[op1, e, esize], unsigned) * Int(Elem[op2, e, esize], unsigned)) + rVal;
    Elem[result, e, esize] = value<(2*esize)-1:esize>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.393 VMULL (integer)

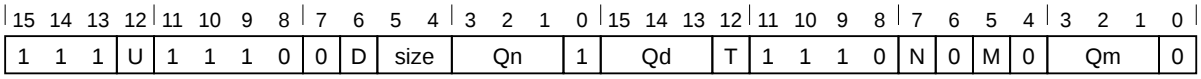
Vector Multiply Long. Performs an element-wise integer multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements. The operation produces a double-width result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMULL variant

```
VMULL<T><v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  unsigned = (U == '1');
9  top      = UInt(T);
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
11 if size == '10' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the source element is used.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- S8 Encoded as size = 00, U = 0
- U8 Encoded as size = 00, U = 1
- S16 Encoded as size = 01, U = 0
- U16 Encoded as size = 01, U = 1
- S32 Encoded as size = 10, U = 0
- U32 Encoded as size = 10, U = 1

<Qd> Destination vector register.

<Qn> Source vector register.

<Qm> Source vector register.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
if esize == 32 then
    op1    = Q[n, UInt(curBeat<1>:T)];
    op2    = Q[m, UInt(curBeat<1>:T)];
    mul    = Int(op1, unsigned) * Int(op2, unsigned);
    result = if curBeat<0> == '1' then mul<63:32> else mul<31:0>;
else
    op1      = Q[n, curBeat];
    op2      = Q[m, curBeat];
    elements = 16 DIV esize;
    for e = 0 to elements-1
        element1 = Elem[op1, e * 2 + top, esize];
        element2 = Elem[op2, e * 2 + top, esize];
        Elem[result, e, esize * 2] = (Int(element1, unsigned) *
                                      Int(element2, unsigned))<2*esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.394 VMULL (polynomial)

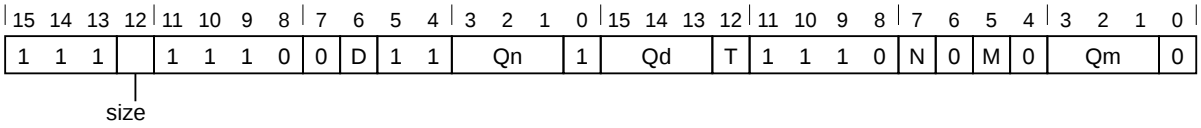
Vector Multiply Long. Performs an element-wise polynomial multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements. The operation produces a double-width result.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMULL variant

```
VMULL<T><v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_Mve);
2 if D == '1' || M == '1' || N == '1' then UNDEFINED;
3 d        = UInt(D:Qd);
4 m        = UInt(M:Qm);
5 n        = UInt(N:Qn);
6 esize    = 8 << UInt(size);
7 elements = 16 DIV esize;
8 top      = UInt(T);
9 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the source element is used.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Specifies whether to do 8x8->16 or 16x16->32 polynomial multiplications.
This parameter must be one of the following values:
- P8 Encoded as size = 0
  Indicates 8x8->16
- P16 Encoded as size = 1
  Indicates 16x16->32

<Qd> Destination vector register.

<Qn> Source vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
```

```
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  op2    = Q[n, curBeat];
9  for e = 0 to elements-1
10     element1 = Elem[op1, e * 2 + top, esize];
11     element2 = Elem[op2, e * 2 + top, esize];
12     Elem[result, e, esize*2] = PolynomialMult(element1, element2)<esize*2-1:0>;
13
14 for e = 0 to 3
15     if elmtMask<e> == '1' then
16         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.395 VMVN (immediate)

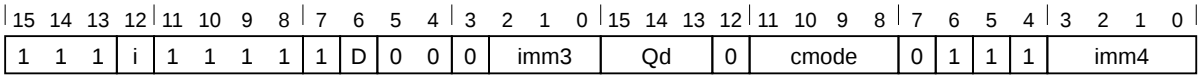
Vector Bitwise NOT. Set each element of a vector register to the bitwise inverse of the immediate operand value. The immediate is generated by the AdvSIMDExpandImm() function based on the requested data type and immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMVN variant

```
VMVN<v>.<dt> Qd, #<imm>
```

### Decode for this encoding

```
1  if cmode == '1110'                  then SEE "VMOV (immediate) (vector)";
2  if cmode IN {'0xx1', 'x0x1', 'xx01'} then SEE "VBIC (immediate)";
3  CheckDecodeFaults(ExtType_Mve);
4  if D     == '1'    then UNDEFINED;
5  if cmode == '11x1' then UNDEFINED;
6  d     = UInt(D:Qd);
7  imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector, for use with the AdvSIMDExpandImm() function.
This parameter must be one of the following values:

- I32 Encoded as:
  - cmode = 0000
  - cmode = 0010
  - cmode = 0100
  - cmode = 0110
  - cmode = 1100
- I16 Encoded as:
  - cmode = 1000
  - cmode = 1010

<Qd> Destination vector register.

<imm> The immediate value to load in to each element. This must be an immediate that can be encoded for use with the AdvSIMDExpandImm() function.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
```

```
6  if curBeat<0> == '0' then
7      result = NOT(imm64<31:0>);
8  else
9      result = NOT(imm64<63:32>);
10
11 for e = 0 to 3
12     if elmtMask<e> == '1' then
13         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.396 VMVN (register)

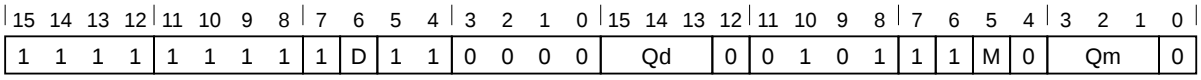
Vector Bitwise Not. Bitwise invert the value of a vector register and place the result in another vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VMVN variant

```
VMVN<v> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = NOT(Q[m, curBeat]);
7
8  for e = 0 to 3
9      if elmtMask<e> == '1' then
10         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.397 VNEG (floating-point)

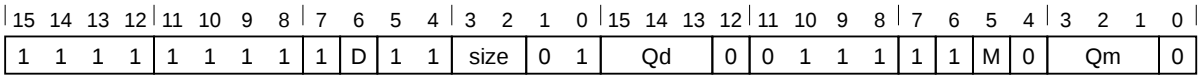
Vector Negate. Negate the value of each element of a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VNEG variant

```
VNEG<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' then UNDEFINED;
3  if size IN {'11', '00'} then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F16 Encoded as size = 01<br>F32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = FPNeg(Elem[op1, e, esize]);
10     Elem[result, e, esize] = value;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.398 VNEG (vector)

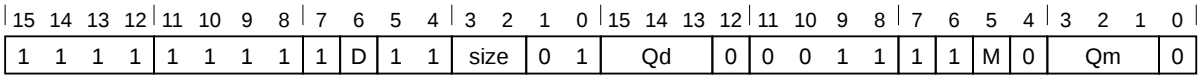
Vector Negate. Negate the value of each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VNEG variant

```
VNEG<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

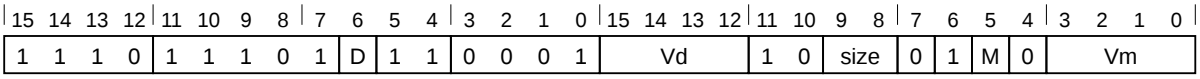
```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = -SInt(Elem[op1, e, esize]);
10     Elem[result, e, esize] = value<esize-1:0>;
11
12 for e = 0 to 3
13     if elmtMask<e> == '1' then
14         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.399 VNEG

Floating-point Negate. Floating-point Negate inverts the sign bit of a half-precision or single-precision or double-precision register, and places the result in the destination register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4      case size of
5          when '01' S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
6          when '10' S[d] = FPNeg(S[m]);
```

```
7           when '11' D[d] = FPNeg(D[m]);
```

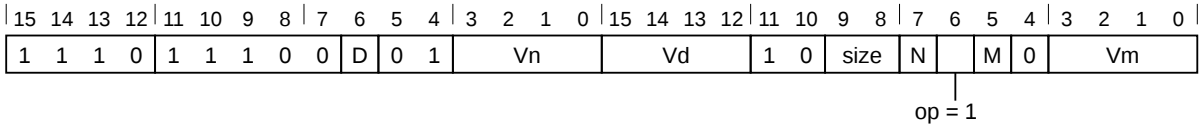
## C2.4.400 VNMLA

Floating-point Multiply Accumulate and Negate. Floating-point Multiply Accumulate and Negate multiplies two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Arm recommends that software does not use the VNMLA instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
    UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
operation = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |

<Dd> Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
<Dn> Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field.
<Dm> Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field.

## Operation for all encodings

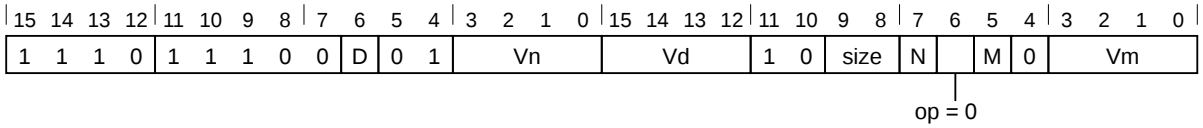
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(
                    product16), TRUE);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16,
                    TRUE);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when '10'
            product32 = FPMul(S[n], S[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, TRUE);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when '11'
            product64 = FPMul(D[n], D[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, TRUE);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

## C2.4.401 VNMLS

Floating-point Multiply Subtract and Negate. Floating-point Multiply Subtract and Negate multiplies two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  operation = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

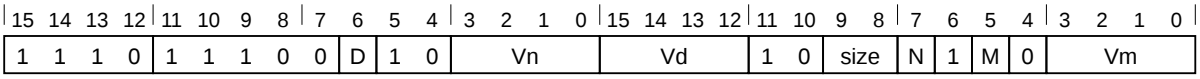
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(
                    product16), TRUE);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16,
                    TRUE);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when '10'
            product32 = FPMul(S[n], S[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, TRUE);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when '11'
            product64 = FPMul(D[n], D[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, TRUE);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

## C2.4.402 VNMUL

Floating-point Multiply and Negate. Floating-point Multiply and Negate multiplies two floating-point register values, and writes the negation of the result to the destination register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VNMUL{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  operation = VFPNegMul_VNMUL;
6  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
7  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
8  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

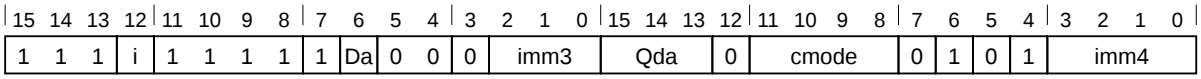
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(
                    product16), TRUE);
                when VFPNegMul_VNMLS  S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16,
                    TRUE);
                when VFPNegMul_VNMUL  S[d] = Zeros(16) : FPNeg(product16);
        when '10'
            product32 = FPMul(S[n], S[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), TRUE);
                when VFPNegMul_VNMLS  S[d] = FPAdd(FPNeg(S[d]), product32, TRUE);
                when VFPNegMul_VNMUL  S[d] = FPNeg(product32);
        when '11'
            product64 = FPMul(D[n], D[m], TRUE);
            case operation of
                when VFPNegMul_VNMLA  D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), TRUE);
                when VFPNegMul_VNMLS  D[d] = FPAdd(FPNeg(D[d]), product64, TRUE);
                when VFPNegMul_VNMUL  D[d] = FPNeg(product64);
```

## C2.4.403 VORN (immediate)

Vector Bitwise OR NOT. This is a pseudo-instruction, equivalent to a VORR (immediate) instruction with the immediate value bitwise inverted.

This is an alias of VORR (immediate).

### T1




**VORN variant**

```
VORN<v>.<dt> Qda, #<imm>
```

**is equivalent to**

```
VORR<v>.<dt> Qda, #~<imm>
```

**and is never the preferred disassembly**

## C2.4.404 VORN

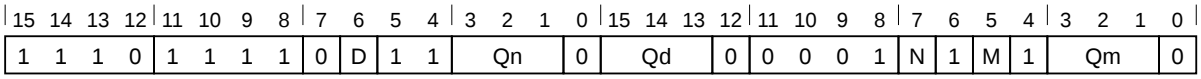
Vector Bitwise Or Not. Compute a bitwise OR NOT of a vector register with another vector register. The result is written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VORN variant

```
VORN<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  n = UInt(N:Qn);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Q[n, curBeat] OR NOT(Q[m, curBeat]);
7
8  for e = 0 to 3
9      if elmtMask<e> == '1' then
10         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.405 VORR (immediate)

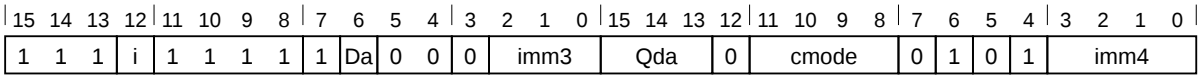
Vector Bitwise OR. OR the value of a vector register with the immediate operand value. The immediate is generated by the AdvSIMDExpandImm() function based on the requested data type and immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VORR variant

```
VORR<v>.<dt> Qda, #<imm>
```

### Decode for this encoding

```
1  if cmode IN {'xxx0', '11xx'} then SEE "VMOV (immediate)";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' then UNDEFINED;
4  da    = UInt(Da:Qda);
5  imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector, for use with the AdvSIMDExpandImm() function.
This parameter must be one of the following values:

I32 Encoded as:
cmode = 0001
cmode = 0011
cmode = 0101
cmode = 0111

I16 Encoded as:
cmode = 1001
cmode = 1011

<Qda> Source and destination vector register.

<imm> The immediate value to load in to each element. This must be an immediate that can be encoded for use with the AdvSIMDExpandImm() function.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  opd    = Q[da, curBeat];
7  imm32  = if curBeat<0> == '0' then imm64<31:0> else imm64<63:32>;
8  result = opd OR imm32;
9
```

```
10  for e = 0 to 3
11      if elmtMask<e> == '1' then
12          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.406 VORR

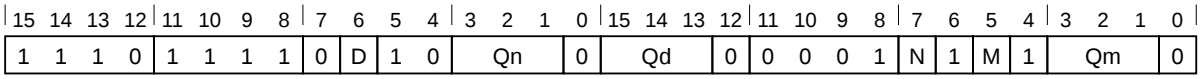
Vector Bitwise Or. Compute a bitwise OR of a vector register with another vector register. The result is written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VORR variant

```
VORR<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  n = UInt(N:Qn);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Q[n, curBeat] OR Q[m, curBeat];
7
8  for e = 0 to 3
9      if elmtMask<e> == '1' then
10         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.407 VPNOT

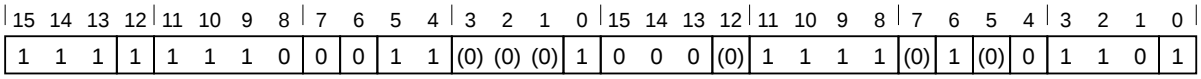
Vector Predicate NOT. Inverts the predicate condition in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | (0) | (0) | (0) | 1 | 0 | 0 | 0 | (0) | 1 | 1 | 1 | 1 | (0) | 1 | (0) | 0 | 1 | 1 | 0 | 1 |

#### T1: VPNOT variant

```
VPNOT<v>
```

### Decode for this encoding

```
1 CheckDecodeFaults(ExtType_Mve);
2
3 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
5
6 Elem[VPR.P0, curBeat, 4] = (NOT Elem[VPR.P0, curBeat, 4]) AND elmtMask;
```

## C2.4.408 VPOP

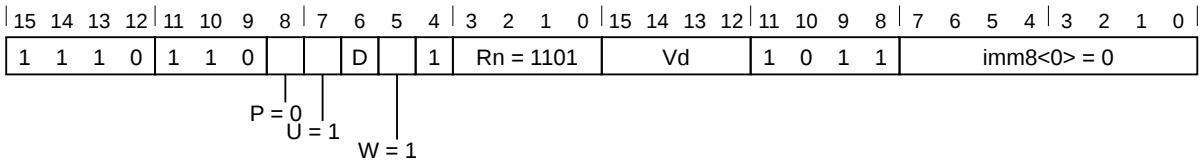
Pop Floating-point registers from stack. Pop Floating-point registers from stack loads multiple consecutive Floating-point registers from the stack.

This instruction is an alias of the VLDM instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM.
- The description of VLDM gives the operational pseudocode for this instruction.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Increment After variant**

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

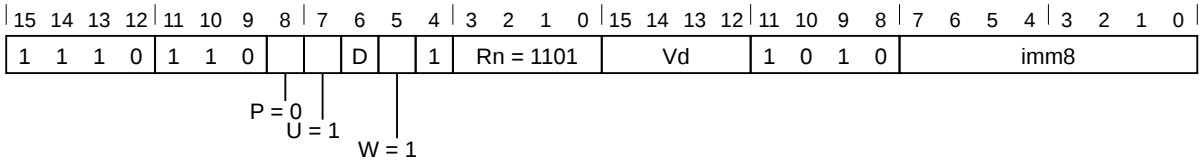
**is equivalent to**

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

**and is always the preferred disassembly.**

### T2

*Armv8-M Floating-point Extension only or MVE*




**Increment After variant**

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

**is equivalent to**

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| `<c>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<q>` | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| `<size>` | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred. |
| `<sreglist>` | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register. |
| `<dreglist>` | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

### Operation for all encodings

The description of VLDM gives the operational pseudocode for this instruction.

## C2.4.409 VPSEL

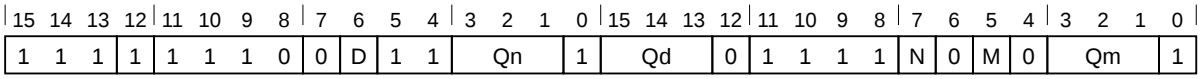
Vector Predicated Select. Compute a bytewise conditional select of a vector register with another vector register, based on the VPR predicate bits

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VPSEL variant

```
VPSEL<v>{.<dt>} Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  n = UInt(N:Qn);
6  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following: S8, S16, S32, U8, U16, U32, I8, I16, I32, F16, F32. |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  opm    = Q[m, curBeat];
8  opn    = Q[n, curBeat];
9  vpr    = Elem[VPR.P0, curBeat, 4];
10 for e = 0 to 3
11     Elem[result, e, 8] = Elem[if vpr<e> == '1' then opn else opm, e, 8];
12
13 for e = 0 to 3
14     if elmtMask<e> == '1' then
15         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.410 VPST

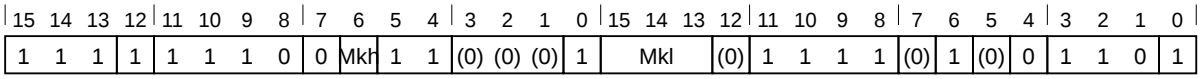
Vector Predicate Set Then. Predicates the following instructions, up to a maximum of four instructions. This instruction is similar to VPT. However no comparison is performed and instead the current value of VPR.P0 is used as the predicate condition.

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VPST variant

```
VPST{x{y{z}}}
```

### Decode for this encoding

```
if Mkh:Mkl == '0000' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);

mask = Mkh:Mkl;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<x> Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into fcond[0], a 1 in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted

<y> Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

<z> Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, -) = GetCurInstrBeat();

// Only one mask field per pair of beats, so the mask is only updated on odd beats.
if curBeat<0> == '1' then
    SetVPTMask(curBeat, mask);
```

## C2.4.411 VPT (floating-point)

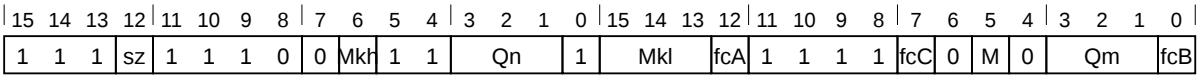
Vector Predicate Then. Predicates the following instructions, up to a maximum of four instructions, by masking the operation of instructions on a per-lane basis based on the VPR.P0 predicate values. The predicated instructions are referred to as the Vector Predication Block or simply the VPT Block. The VPR.P0 predicate values may be inverted after each instruction in the VPT block based on the mask fields (see x, y, and z).

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VPT variant

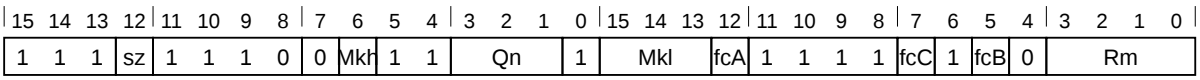
```
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if (fcA == '0' && fcB == '1') || Mkh:Mkl == '0000' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_MveFp);
3  if M == '1' then UNDEFINED;
4  m          = UInt(M:Qm);
5  n          = UInt(Qn);
6  mask       = Mkh:Mkl;
7  fcond      = fcA:fcB:fcC;
8  withScalar = FALSE;
9  esize      = 8 << UInt(if sz == '1' then '01' else '10');
10 elements   = 32 DIV esize;
11 ebytes     = esize DIV 8;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VPT variant

```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
```

### Decode for this encoding

```
1  if Mkh:Mkl == '0000' then SEE "Related encodings";
2  if Rm      == '1101' then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_MveFp);
4  m          = UInt(Rm);
5  n          = UInt(Qn);
6  mask       = Mkh:Mkl;
7  fcond      = fcA:fcB:fcC;
8  withScalar = TRUE;
9  esize      = 8 << UInt(if sz == '1' then '01' else '10');
10 elements   = 32 DIV esize;
```

```
ebytes     = esize DIV 8;
if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
if fcA == '0' && fcB == '1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

<dt> Size: indicates the floating-point format used.
This parameter must be one of the following values:
- F32 Encoded as sz = 0
- F16 Encoded as sz = 1

<fc> The comparison condition to use.
This parameter must be one of the following values:
- EQ Encoded as fcA = 0, fcB = 0, fcC = 0
- NE Encoded as fcA = 0, fcB = 0, fcC = 1
- GE Encoded as fcA = 1, fcB = 0, fcC = 0
- LT Encoded as fcA = 1, fcB = 0, fcC = 1
- GT Encoded as fcA = 1, fcB = 1, fcC = 0
- LE Encoded as fcA = 1, fcB = 1, fcC = 1

<Qn> Source vector register.

<Qm> Source vector register.

<Rm> Source general-purpose register (ZR is permitted, PC is not).

<x> Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into fcond[0], a 1 in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted

<y> Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

<z> Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1      = Q[n, curBeat];
beatPred = Zeros(4);
if withScalar then
    op2 = RZ[m]<esize-1:0>;
else
    opm = Q[m, curBeat];
for e = 0 to elements-1
    if !withScalar then
        op2 = Elem[opm, e, esize];
    (flN, flZ, flC, flV) = FPCompare(Elem[op1, e, esize], op2, TRUE, TRUE);
    pred = ConditionHolds(fcond, flN, flZ, flC, flV);
    Elem[beatPred, e, ebytes] = Replicate(if pred then '1' else '0');

Elem[VPR.P0, curBeat, 4] = beatPred AND elmtMask;
// Only one mask field per pair of beats, so the mask is only updated on odd beats.
if curBeat<0> == '1' then
    SetVPTMask(curBeat, mask);
```

## C2.4.412 VPT

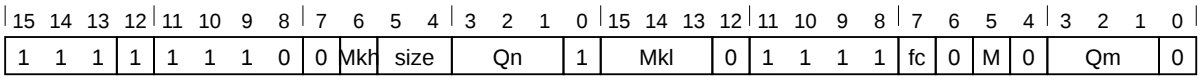
Vector Predicate Then. Predicates the following instructions, up to a maximum of four instructions, by masking the operation of instructions on a per-lane basis based on the VPR.P0 predicate values. The predicated instructions are referred to as the Vector Predication Block or simply the VPT Block. The VPR.P0 predicate values may be inverted after each instruction in the VPT block based on the mask fields (see x, y, and z).

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VPT variant

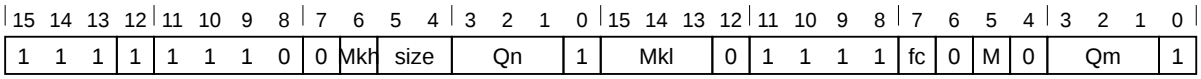
```
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

### Decode for this encoding

```
1  if Mkh:Mkl == '0000' then SEE "Related encodings";
2  if size    == '11'   then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  if M == '1' then UNDEFINED;
5  m           = UInt(M:Qm);
6  n           = UInt(Qn);
7  mask        = Mkh:Mkl;
8  fcond       = '00':fc;
9  withScalar  = FALSE;
10 esize       = 8 << UInt(size);
11 elements    = 32 DIV esize;
12 ebytes      = esize DIV 8;
13 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VPT variant

```
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

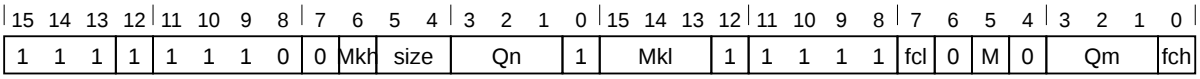
### Decode for this encoding

```
1  if Mkh:Mkl == '0000' then SEE "Related encodings";
2  if size    == '11'   then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  if M == '1' then UNDEFINED;
5  m           = UInt(M:Qm);
6  n           = UInt(Qn);
7  mask        = Mkh:Mkl;
8  fcond       = '01':fc;
9  withScalar  = FALSE;
```

```
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
ebytes     = esize DIV 8;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M MVE*




### T3: VPT variant

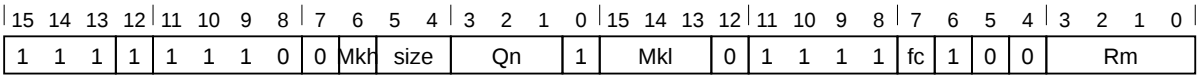
```
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

## Decode for this encoding

```
if Mkh:Mkl == '0000' then SEE "Related encodings";
if size    == '11'   then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if M == '1' then UNDEFINED;
m          = UInt(M:Qm);
n          = UInt(Qn);
mask       = Mkh:Mkl;
fcond      = '1':fch:fcl;
withScalar = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
ebytes     = esize DIV 8;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M MVE*




### T4: VPT variant

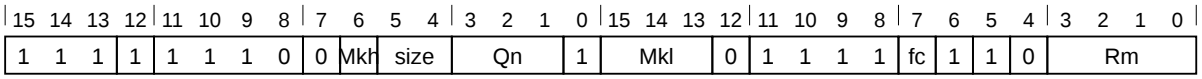
```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
```

## Decode for this encoding

```
if Mkh:Mkl == '0000' then SEE "Related encodings";
if size    == '11'   then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
m          = UInt(Rm);
n          = UInt(Qn);
mask       = Mkh:Mkl;
fcond      = '00':fc;
withScalar = TRUE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
ebytes     = esize DIV 8;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## T5

*Armv8.1-M MVE*




### T5: VPT variant

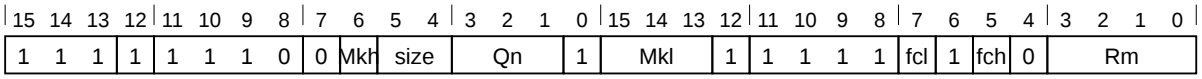
```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
```

## Decode for this encoding

```
1  if Mkh:Mkl == '0000' then SEE "Related encodings";
2  if size    == '11'   then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  m          = UInt(Rm);
5  n          = UInt(Qn);
6  mask       = Mkh:Mkl;
7  fcond      = '01':fc;
8  withScalar = TRUE;
9  esize      = 8 << UInt(size);
10 elements   = 32 DIV esize;
11 ebytes     = esize DIV 8;
12 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
13 if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*




### T6: VPT variant

```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
```

## Decode for this encoding

```
1  if Mkh:Mkl == '0000' then SEE "Related encodings";
2  if size    == '11'   then SEE "Related encodings";
3  CheckDecodeFaults(ExtType_Mve);
4  m          = UInt(Rm);
5  n          = UInt(Qn);
6  mask       = Mkh:Mkl;
7  fcond      = '1':fch:fcl;
8  withScalar = TRUE;
9  esize      = 8 << UInt(size);
10 elements   = 32 DIV esize;
11 ebytes     = esize DIV 8;
12 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
13 if Rm == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

| | |
|---|---|
| `<dt>` | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>`I8` Encoded as `size = 00`<br>`I16` Encoded as `size = 01`<br>`I32` Encoded as `size = 10` |
| `<fc>` | The comparison condition to use. |

This parameter must be one of the following values:

- `EQ` Encoded as `fc = 0`
- `NE` Encoded as `fc = 1`

### Assembler symbols for T2 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `U8` Encoded as `size = 00`
- `U16` Encoded as `size = 01`
- `U32` Encoded as `size = 10`

`<fc>` The comparison condition to use.
This parameter must be one of the following values:

- `CS` Encoded as `fc = 0`
- `HI` Encoded as `fc = 1`

### Assembler symbols for T3 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `S8` Encoded as `size = 00`
- `S16` Encoded as `size = 01`
- `S32` Encoded as `size = 10`

`<fc>` The comparison condition to use.
This parameter must be one of the following values:

- `GE` Encoded as `fch = 0, fcl = 0`
- `LT` Encoded as `fch = 0, fcl = 1`
- `GT` Encoded as `fch = 1, fcl = 0`
- `LE` Encoded as `fch = 1, fcl = 1`

### Assembler symbols for T4 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `I8` Encoded as `size = 00`
- `I16` Encoded as `size = 01`
- `I32` Encoded as `size = 10`

`<fc>` The comparison condition to use.
This parameter must be one of the following values:

- `EQ` Encoded as `fc = 0`
- `NE` Encoded as `fc = 1`

### Assembler symbols for T5 encodings

`<dt>` Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

- `U8` Encoded as `size = 00`
- `U16` Encoded as `size = 01`
- `U32` Encoded as `size = 10`

`<fc>` The comparison condition to use.
This parameter must be one of the following values:

- `CS` Encoded as `fc = 0`

HI Encoded as fc = 1

## Assembler symbols for T6 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00 |
| S16 | Encoded as | size = 01 |
| S32 | Encoded as | size = 10 |

<fc> The comparison condition to use.
This parameter must be one of the following values:

| | | |
|---|---|---|
| GE | Encoded as | fch = 0, fcl = 0 |
| LT | Encoded as | fch = 0, fcl = 1 |
| GT | Encoded as | fch = 1, fcl = 0 |
| LE | Encoded as | fch = 1, fcl = 1 |

## Assembler symbols for all encodings

<Qn> Source vector register.
<Qm> Source vector register.
<Rm> Source general-purpose register (ZR is permitted, PC is not).
<x> Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into fcond[0], a 1 in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted
<y> Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.
<z> Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1      = Q[n, curBeat];
beatPred = Zeros(4);
if withScalar then
    op2 = RZ[m]<esize-1:0>;
else
    opm = Q[m, curBeat];
for e = 0 to elements-1
    if !withScalar then
        op2 = Elem[opm, e, esize];
    (result, flC, flV) = AddWithCarry(Elem[op1, e, esize], NOT(op2), '1');
    flN                = result<esize-1>;
    flZ                = IsZeroBit(result);
    pred = ConditionHolds(fcond, flN, flZ, flC, flV);
    Elem[beatPred, e, ebytes] = Replicate(if pred then '1' else '0');

Elem[VPR.P0, curBeat, 4] = beatPred AND elmtMask;
// Only one mask field per pair of beats, so the mask is only updated on odd beats.
if curBeat<0> == '1' then
    SetVPTMask(curBeat, mask);
```

## C2.4.413 VPUSH

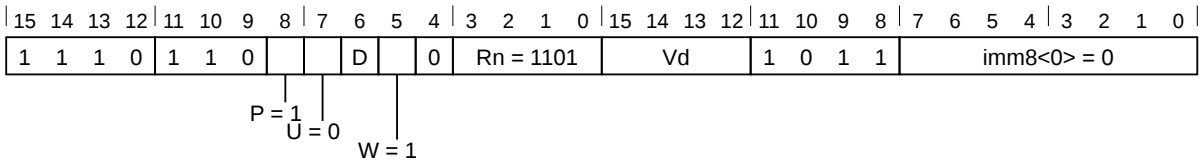
Push Floating-point registers to stack. Push Floating-point registers to stack stores multiple consecutive registers from the Floating-point register file to the stack.

This instruction is an alias of the VSTM instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM.
- The description of VSTM gives the operational pseudocode for this instruction.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

```
VPUSH{<c>}{<q>}{.<size>} <dreglist>
```

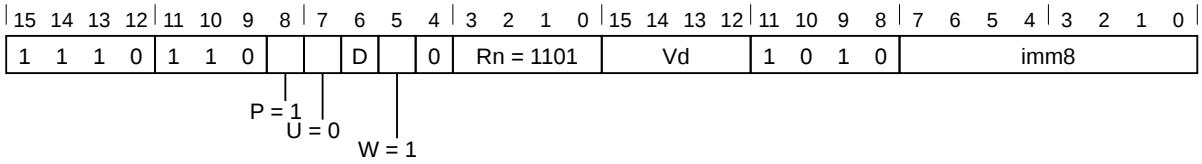
**is equivalent to**

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>
```

**and is always the preferred disassembly.**

### T2

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

```
VPUSH{<c>}{<q>}{.<size>} <sreglist>
```

**is equivalent to**

```
VSTMDB{<c>}{<q>}{.<size>} SP!, <sreglist>
```

**and is always the preferred disassembly.**

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred. |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register. |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

### Operation for all encodings

The description of VSTM gives the operational pseudocode for this instruction.

## C2.4.414 VQABS

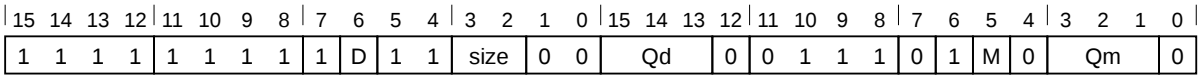
Vector Saturating Absolute. Compute the absolute value of and saturate each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQABS variant

`VQABS<v>.<dt> Qd, Qm`

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = Abs(SInt(Elem[op1, e, esize]));
10     (Elem[result, e, esize], sat) = SignedSatQ(value, esize);
11     if sat && elmtMask<e*(esize>>3)> == '1' then
12         FPSCR.QC = '1';
13
14 for e = 0 to 3
15     if elmtMask<e> == '1' then
16         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.415 VQADD

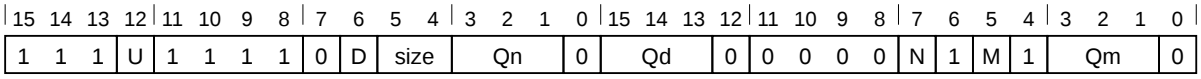
Vector Saturating Add. Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is saturated before being written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQADD variant

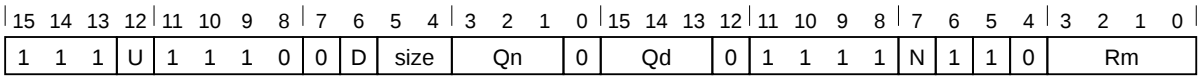
```
VQADD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQADD variant

```
VQADD<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = TRUE;
11 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Size: indicates the size of the elements in the vector.<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if !withScalar then
    op2 = Q[m, curBeat];
for e = 0 to elements-1
    if withScalar then
        value = Int(Elem[op1, e, esize], unsigned) + Int(R[m]<esize-1:0>, unsigned);
    else
        value = Int(Elem[op1, e, esize], unsigned) + Int(Elem[op2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(value, esize, unsigned);
    if sat && elmtMask<e*(esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.416 VQDMLADH, VQRDMLADH

Vector Saturating Doubling Multiply Add Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Add Dual Returning High Half. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together and doubling the result. The high halves of the resulting values are selected as the final results. The base variant writes the results into the lower element of each pair of elements in the destination register, whereas the exchange variant writes to the upper element in each pair. The results are optionally rounded before the high half is selected and saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | D | size | | Qn | | | 0 | Qd | | | X | 1 | 1 | 1 | 0 | N | 0 | M | 0 | Qm | | | 0 |

#### T1: VQDMLADH variant

```
VQDMLADH{X}<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  exchange = (X == '1');
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 round    = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
12 if size == '10' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | D | size | | Qn | | | 0 | Qd | | | X | 1 | 1 | 1 | 0 | N | 0 | M | 0 | Qm | | | 1 |

#### T2: VQRDMLADH variant

```
VQRDMLADH{X}<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
```

```
m        = UInt(M:Qm);
n        = UInt(N:Qn);
exchange = (X == '1');
esize    = 8 << UInt(size);
elements = 32 DIV esize;
round    = TRUE;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
if size == '10' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <X> | Exchange adjacent pairs of values in Qm.<br>This parameter must be one of the following values:<br>" Encoded as X = 0<br>X Encoded as X = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
rVal   = if round then 1 << (esize-1) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if (curBeat<0> == '1') == exchange then
        if exchange then
            mul1 = SInt(Q[n, curBeat])   * SInt(Q[m, curBeat-1]);
            mul2 = SInt(Q[n, curBeat-1]) * SInt(Q[m, curBeat]);
        else
            mul1 = SInt(Q[n, curBeat])   * SInt(Q[m, curBeat]);
            mul2 = SInt(Q[n, curBeat+1]) * SInt(Q[m, curBeat+1]);
        (value, sat) = SignedSatQ(2 * (mul1 + mul2) + rVal, esize*2);
        result = value<63:32>;
        if sat && elmtMask<0> == '1' then
            FPSCR.QC = '1';
    else
        // No computation on this beat, so don't write to the dest register.
        elmtMask = Zeros();
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if (e<0> == '1') == exchange then
            if exchange then
                mul1 = SInt(Elem[op1, e,   esize]) * SInt(Elem[op2, e-1, esize]);
                mul2 = SInt(Elem[op1, e-1, esize]) * SInt(Elem[op2, e,   esize]);
            else
                mul1 = SInt(Elem[op1, e,   esize]) * SInt(Elem[op2, e,   esize]);
                mul2 = SInt(Elem[op1, e+1, esize]) * SInt(Elem[op2, e+1, esize]);
            (value, sat) = SignedSatQ(2 * (mul1 + mul2) + rVal, esize*2);
```

```
            Elem[result, e, esize] = value<esize+esize-1:esize>;
            if sat && elmtMask<e*(esize>>3)> == '1' then
                FPSCR.QC = '1';
        else
            // No computation on this lane, so don't write to the dest register.
            Elem[elmtMask, e, esize DIV 8] = Zeros();

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.417 VQDMLAH, VQRDMLAH (vector by scalar plus vector)

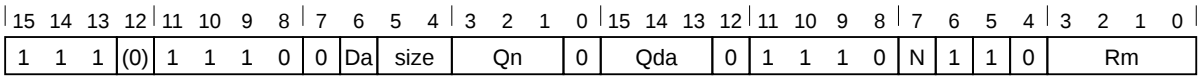
Vector Saturating Doubling Multiply Accumulate, Vector Saturating Rounding Doubling Multiply Accumulate. Multiply each element in the source vector by a scalar value, double the result and add to the respective element from the destination vector High Half. Store the high half of each result in the destination register. The result is optionally rounded before the high half is selected and saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQDMLAH variant

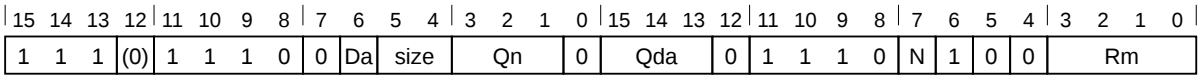
```
VQDMLAH<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if Da == '1' || N == '1' then UNDEFINED;
da       = UInt(Da:Qda);
m        = UInt(Rm);
n        = UInt(N:Qn);
unsigned = FALSE;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
round    = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQRDMLAH variant

```
VQRDMLAH<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if Da == '1' || N == '1' then UNDEFINED;
da       = UInt(Da:Qda);
m        = UInt(Rm);
n        = UInt(N:Qn);
unsigned = FALSE;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
round    = TRUE;
```

```
11  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12  if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qda> | Accumulator vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3
4   (curBeat, elmtMask) = GetCurInstrBeat();
5
6   result   = Zeros(32);
7   op1      = Q[n, curBeat];
8   element2 = Int(R[m]<esize-1:0>, unsigned);
9   op3      = Q[da, curBeat];
10  rVal     = if round then 1 << (esize-1) else 0;
11  for e = 0 to elements-1
12      element1 = Int(Elem[op1, e, esize], unsigned);
13      element3 = Int(Elem[op3, e, esize], unsigned) << esize;
14      (value, sat) = SatQ((2 * element1 * element2) + element3 + rVal, esize*2, unsigned);
15      Elem[result, e, esize] = value<esize+esize-1:esize>;
16      if sat && elmtMask<e*(esize>>3)> == '1' then
17          FPSCR.QC = '1';
18
19  for e = 0 to 3
20      if elmtMask<e> == '1' then
21          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.418 VQDMLASH, VQRDMLASH (vector by vector plus scalar)

Vector Saturating Doubling Multiply Accumulate Scalar High Half, Vector Saturating Rounding Doubling Multiply Accumulate Scalar High Half. Multiply each element in the source vector by the respective element from the destination vector, double the result and add to a scalar value. Store the high half of each result in the destination register. The result is optionally rounded before the high half is selected and saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | (0) | 1 | 1 | 1 | 0 | 0 | Da | size | | Qn | | | 0 | Qda | | | 1 | 1 | 1 | 1 | 0 | N | 1 | 1 | 0 | Rm | | | |

#### T1: VQDMLASH variant

```
VQDMLASH<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || N == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(Rm);
6  n        = UInt(N:Qn);
7  unsigned = FALSE;
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 round    = FALSE;
11 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | (0) | 1 | 1 | 1 | 0 | 0 | Da | size | | Qn | | | 0 | Qda | | | 1 | 1 | 1 | 1 | 0 | N | 1 | 0 | 0 | Rm | | | |

#### T2: VQRDMLASH variant

```
VQRDMLASH<v>.<dt> Qda, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' || N == '1' then UNDEFINED;
4  da       = UInt(Da:Qda);
5  m        = UInt(Rm);
6  n        = UInt(N:Qn);
7  unsigned = FALSE;
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 round    = TRUE;
```

```
11  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12  if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qda> | Source and destination vector register. |
| <Qn> | Source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3
4   (curBeat, elmtMask) = GetCurInstrBeat();
5
6   result   = Zeros(32);
7   op1      = Q[n, curBeat];
8   op2      = Q[da, curBeat];
9   element3 = Int(R[m]<esize-1:0>, unsigned) << esize;
10  rVal     = if round then 1 << (esize-1) else 0;
11  for e = 0 to elements-1
12      element1 = Int(Elem[op1, e, esize], unsigned);
13      element2 = Int(Elem[op2, e, esize], unsigned);
14      (value, sat) = SatQ((2 * element1 * element2) + element3 + rVal, esize*2, unsigned);
15      Elem[result, e, esize] = value<esize+esize-1:esize>;
16      if sat && elmtMask<e*(esize>>3)> == '1' then
17          FPSCR.QC = '1';
18
19  for e = 0 to 3
20      if elmtMask<e> == '1' then
21          Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.419 VQDMLSDH, VQRDMLSDH

Vector Saturating Doubling Multiply Subtract Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Subtract Dual Returning High Half. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other and doubling the result. The high halves of the resulting values are selected as the final results. The base variant writes the results into the lower element of each pair of elements in the destination register, whereas the exchange variant writes to the upper element in each pair. The results are optionally rounded before the high half is selected and saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | D | size | | Qn | | | 0 | Qd | | | X | 1 | 1 | 1 | 0 | N | 0 | M | 0 | Qm | | | 0 |

#### T1: VQDMLSDH variant

```
VQDMLSDH{X}<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  exchange = (X == '1');
8  esize    = 8 << UInt(size);
9  elements = 32 DIV esize;
10 round    = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
12 if size == '10' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | D | size | | Qn | | | 0 | Qd | | | X | 1 | 1 | 1 | 0 | N | 0 | M | 0 | Qm | | | 1 |

#### T2: VQRDMLSDH variant

```
VQRDMLSDH{X}<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
```

```
m        = UInt(M:Qm);
n        = UInt(N:Qn);
exchange = (X == '1');
esize    = 8 << UInt(size);
elements = 32 DIV esize;
round    = TRUE;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
if size == '10' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <X> | Exchange adjacent pairs of values in Qm.<br>This parameter must be one of the following values:<br>" Encoded as X = 0<br>X Encoded as X = 1 |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
rVal   = if round then 1 << (esize-1) else 0;
// 32 bit operations are handled differently as they perform cross beat
// register accesses
if esize == 32 then
    if (curBeat<0> == '1') == exchange then
        if exchange then
            mul1 = SInt(Q[n, curBeat])   * SInt(Q[m, curBeat-1]);
            mul2 = SInt(Q[n, curBeat-1]) * SInt(Q[m, curBeat]);
        else
            mul1 = SInt(Q[n, curBeat])   * SInt(Q[m, curBeat]);
            mul2 = SInt(Q[n, curBeat+1]) * SInt(Q[m, curBeat+1]);
        (value, sat) = SignedSatQ(2 * (mul1 - mul2) + rVal, esize*2);
        result = value<63:32>;
        if sat && elmtMask<0> == '1' then
            FPSCR.QC = '1';
    else
        // No computation on this beat, so don't write to the dest register.
        elmtMask = Zeros();
else
    op1 = Q[n, curBeat];
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        if (e<0> == '1') == exchange then
            if exchange then
                mul1 = SInt(Elem[op1, e,   esize]) * SInt(Elem[op2, e-1, esize]);
                mul2 = SInt(Elem[op1, e-1, esize]) * SInt(Elem[op2, e,   esize]);
            else
                mul1 = SInt(Elem[op1, e,   esize]) * SInt(Elem[op2, e,   esize]);
                mul2 = SInt(Elem[op1, e+1, esize]) * SInt(Elem[op2, e+1, esize]);
            (value, sat) = SignedSatQ(2 * (mul1 - mul2) + rVal, esize*2);
```

```
            Elem[result, e, esize] = value<esize+esize-1:esize>;
            if sat && elmtMask<e*(esize>>3)> == '1' then
                FPSCR.QC = '1';
        else
            // No computation on this lane, so don't write to the dest register.
            Elem[elmtMask, e, esize DIV 8] = Zeros();

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.420 VQDMULH, VQRDMULH

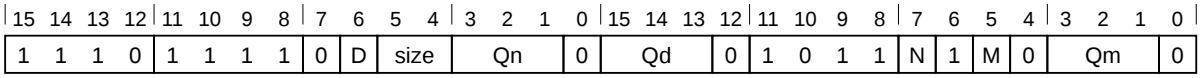
Vector Saturating Doubling Multiply Returning High Half, Vector Saturating Rounding Doubling Multiply Returning High Half. Multiply a general-purpose register value by each element of a vector register to produce a vector of results or multiply each element of a vector register by its corresponding element in another vector register, double the results, and place the most significant half of the final results in the destination vector. The results are optionally rounded before being saturated.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQDMULH variant

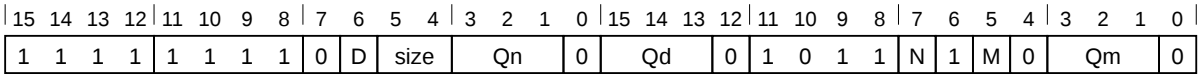
```
VQDMULH<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  withScalar = FALSE;
10 round      = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQRDMULH variant

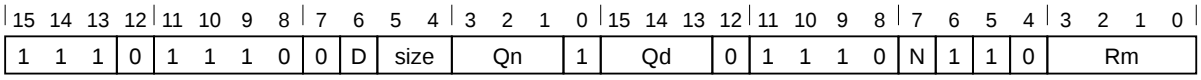
```
VQRDMULH<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  withScalar = FALSE;
10 round      = TRUE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T3

*Armv8.1-M MVE*




#### T3: VQDMULH variant

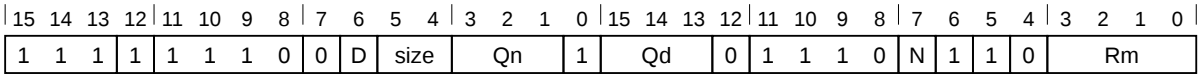
`VQDMULH<v>.<dt> Qd, Qn, Rm`

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || N == '1' then UNDEFINED;
d          = UInt(D:Qd);
m          = UInt(Rm);
n          = UInt(N:Qn);
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
withScalar = TRUE;
round      = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T4

*Armv8.1-M MVE*




#### T4: VQRDMULH variant

`VQRDMULH<v>.<dt> Qd, Qn, Rm`

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || N == '1' then UNDEFINED;
d          = UInt(D:Qd);
m          = UInt(Rm);
n          = UInt(N:Qn);
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
withScalar = TRUE;
round      = TRUE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

| | |
|---|---|
| <Qn> | First source vector register. |

### Assembler symbols for T2 encodings

| | |
|---|---|
| <Qn> | First source vector register. |

### Assembler symbols for T3 encodings

| | |
|---|---|
| <Qn> | Source vector register. |

### Assembler symbols for T4 encodings

| | |
|---|---|
| <Qn> | Source vector register. |

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
op2    = if withScalar then R[m]           else Q[m, curBeat];
rVal   = if round      then 1 << (esize-1) else 0;
for e = 0 to elements-1
    opm   = if withScalar then op2<esize-1:0> else Elem[op2, e, esize];
    value = ((2 * SInt(Elem[op1, e, esize]) * SInt(opm)) + rVal) >> esize;
    (Elem[result, e, esize], sat) = SignedSatQ(value, esize);
    if sat && elmtMask<e*(esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.421 VQDMULL

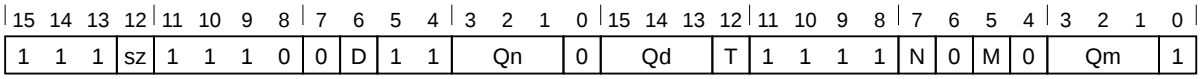
Vector Multiply Long. Performs an element-wise integer multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements or the lower single-width portion of the general-purpose register. The product of the multiplication is doubled and saturated to produce a double-width product that is written back to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQDMULL variant

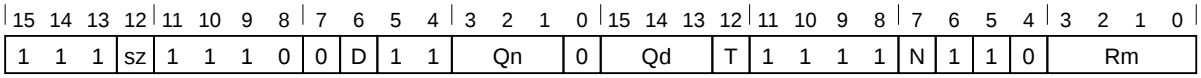
```
VQDMULL<T><v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(M:Qm);
5  n          = UInt(N:Qn);
6  esize      = 16 << UInt(sz);
7  top        = UInt(T);
8  withScalar = FALSE;
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
10 if sz == '1' && (D:Qd == M:Qm || D:Qd == N:Qn) then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQDMULL variant

```
VQDMULL<T><v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(Rm);
5  n          = UInt(N:Qn);
6  esize      = 16 << UInt(sz);
7  top        = UInt(T);
8  withScalar = TRUE;
9  if InITBlock()               then CONSTRAINED_UNPREDICTABLE;
10 if D:Qd == N:Qn && sz == '1' then CONSTRAINED_UNPREDICTABLE;
11 if Rm   == '11x1'            then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <T> | Specifies which half of the source element is used.<br>This parameter must be one of the following values:<br>B Encoded as T = 0<br>Indicates bottom half<br>T Encoded as T = 1<br>Indicates top half |
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S16 Encoded as sz = 0<br>S32 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | Source vector register. |
| <Qm> | Source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
if esize == 32 then
    op1 = Q[n, UInt(curBeat<1>:T)];
    if withScalar then
        op2 = R[m];
    else
        op2 = Q[m, UInt(curBeat<1>:T)];
    (mul, sat) = SignedSatQ(2 * SInt(op1) * SInt(op2), 64);
    result     = if curBeat<0> == '1' then mul<63:32> else mul<31:0>;
    if sat && elmtMask<0> == '1' then
        FPSCR.QC = '1';
else
    op1 = Q[n, curBeat];
    if withScalar then
        op2 = R[m];
    else
        op2 = Q[m, curBeat];
    elements = 16 DIV esize;
    for e = 0 to elements-1
        element1 = Elem[op1, e * 2 + top, esize];
        if withScalar then
            element2 = Elem[op2, 0, esize];
        else
            element2 = Elem[op2, e * 2 + top, esize];
        value = 2 * SInt(element1) * SInt(element2);
        (Elem[result, e, esize * 2], sat) = SignedSatQ(value, 2 * esize);
        if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
            FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.422 VQMOVN

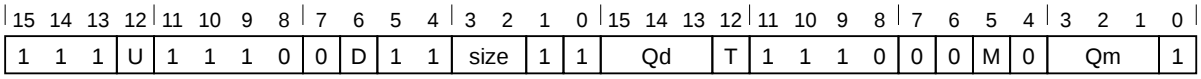
Vector Saturating Move and Narrow. Performs an element-wise saturation to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQMOVN variant

```
VQMOVN<T><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D    == '1' || M == '1' then UNDEFINED;
4  if size == '10'            then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(M:Qm);
7  unsigned = (U == '1');
8  esize    = 8 << UInt(size);
9  elements = 16 DIV esize;
10 top      = UInt(T);
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as size = 00, U = 0
- U16 Encoded as size = 00, U = 1
- S32 Encoded as size = 01, U = 0
- U32 Encoded as size = 01, U = 1

<Qd> Destination vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
```

```
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  op1    = Q[m, curBeat];
7  result = Q[d, curBeat];
8  for e = 0 to elements-1
9      operand = Int(Elem[op1, e, 2*esize], unsigned);
10     (value, sat) = SatQ(operand, esize, unsigned);
11     Elem[result, 2*e + top, esize] = value;
12     if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
13         FPSCR.QC = '1';
14
15 for e = 0 to 3
16     if elmtMask<e> == '1' then
17         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.423 VQMOVUN

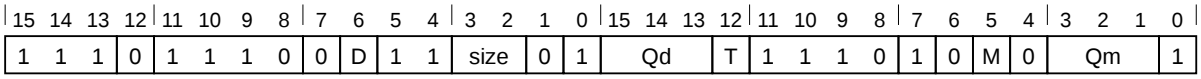
Vector Saturating Move Unsigned and Narrow. Performs an element-wise saturation to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value. The result is always saturated to an unsigned value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQMOVUN variant

```
VQMOVUN<T><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D    == '1' || M == '1' then UNDEFINED;
4  if size == '10'            then UNDEFINED;
5  d = UInt(D:Qd);
6  m = UInt(M:Qm);
7  unsigned     = FALSE;
8  destUnsigned = TRUE;
9  esize        = 8 << UInt(size);
10 elements     = 16 DIV esize;
11 top          = UInt(T);
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as size = 00
- S32 Encoded as size = 01

<Qd> Destination vector register.

<Qm> Source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
```

```
op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, 2*esize], unsigned);
    (value, sat) = SatQ(operand, esize, destUnsigned);
    Elem[result, 2*e + top, esize] = value;
    if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.424 VQNEG

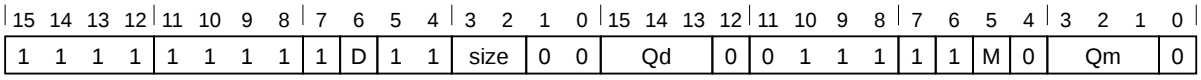
Vector Saturating Negate. Negate the value and saturate each element in a vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQNEG variant

```
VQNEG<v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  esize    = 8 << UInt(size);
7  elements = 32 DIV esize;
8  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00<br>S16 Encoded as size = 01<br>S32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  op1    = Q[m, curBeat];
8  for e = 0 to elements-1
9      value = -SInt(Elem[op1, e, esize]);
10     (Elem[result, e, esize], sat) = SignedSatQ(value, esize);
11     if sat && elmtMask<e*(esize>>3)> == '1' then
12         FPSCR.QC = '1';
13
14 for e = 0 to 3
15     if elmtMask<e> == '1' then
16         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.425 VQRSHL

Vector Saturating Rounding Shift Left. The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

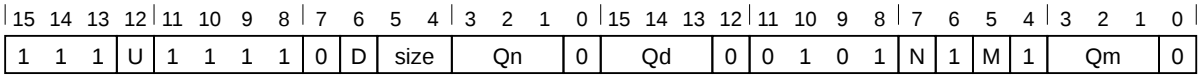
The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQRSHL variant

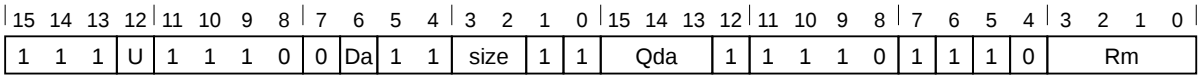
```
VQRSHL<v>.<dt> Qd, Qm, Qn
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  unsigned   = (U == '1');
8  withScalar = FALSE;
9  withVector = TRUE;
10 esize      = 8 << UInt(size);
11 elements   = 32 DIV esize;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQRSHL variant

```
VQRSHL<v>.<dt> Qda, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' then UNDEFINED;
4  da         = UInt(Da:Qda);
5  m          = UInt(Rm);
6  d          = da;
7  n          = m;
8  m          = da;
```

```
unsigned   = (U == '1');
withScalar = TRUE;
withVector = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register, the elements of which containing the amount to shift by. |
| <Rm> | Source general-purpose register containing the amount to shift by. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
if withVector then
    op2 = Q[n, curBeat];
if withScalar then
    shiftAmount = SInt(R[n]<7:0>);
for e = 0 to elements-1
    if withVector then
        shiftAmount = SInt(Elem[op2, e, esize]<7:0>);
    // 0 for left shift, 2^(n-1) for right shift
    round_const  = 1 << (-1-shiftAmount);
    operand      = Int(Elem[op1, e, esize], unsigned);
    (value, sat) = SatQ((operand + round_const) << shiftAmount, esize, unsigned);
    Elem[result, e, esize] = value;
    if sat && elmtMask<e*(esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.426 VQRSHRN

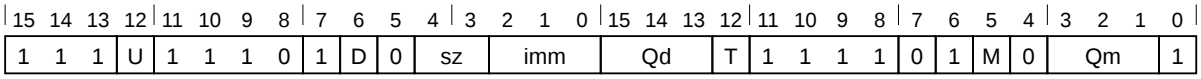
Vector Saturating Rounding Shift Right and Narrow. Performs an element-wise saturation to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQRSHRN variant

```
VQRSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d        = UInt(D:Qd);
4  m        = UInt(M:Qm);
5  unsigned = (U == '1');
6  imm5     = sz:imm;
7  case imm5 of
8      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
9      when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
10     otherwise UNDEFINED;
11 esize    = 8 << UInt(size);
12 elements = 16 DIV esize;
13 top      = UInt(T);
14 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– The size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as sz = 01, U = 0
- U16 Encoded as sz = 01, U = 1
- S32 Encoded as sz = 1x, U = 0
- U32 Encoded as sz = 1x, U = 1

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, 2*esize], unsigned);
    // 0 for left shift, 2^(n-1) for right shift
    operand = operand + (1 << (shiftAmount-1));
    operand = operand >> shiftAmount;
    (value, sat) = SatQ(operand, esize, unsigned);
    Elem[result, 2*e + top, esize] = value;
    if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.427 VQRSHRUN

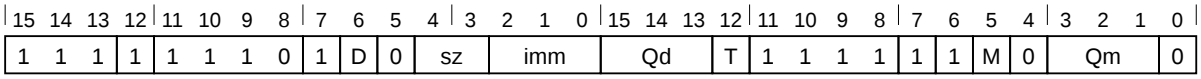
Vector Saturating Rounding Shift Right Unsigned and Narrow. Performs an element-wise saturation to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQRSHRUN variant

```
VQRSHRUN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  unsigned     = FALSE;
6  destUnsigned = TRUE;
7  imm5         = sz:imm;
8  case imm5 of
9      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
10     when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
11     otherwise UNDEFINED;
12 esize    = 8 << UInt(size);
13 elements = 16 DIV esize;
14 top      = UInt(T);
15 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> The size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as sz = 01
- S32 Encoded as sz = 1x

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, 2*esize], unsigned);
    // 0 for left shift, 2^(n-1) for right shift
    operand = operand + (1 << (shiftAmount-1));
    operand = operand >> shiftAmount;
    (value, sat) = SatQ(operand, esize, destUnsigned);
    Elem[result, 2*e + top, esize] = value;
    if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.428 VQSHL, VQSHLU

Vector Saturating Shift Left, Vector Saturating Shift Left Unsigned. The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

The immediate variant shifts each element of a vector register to the left by the immediate value.

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

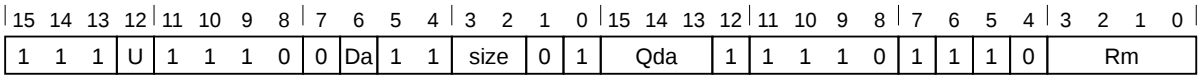
The unsigned variant produces unsigned results, although the operands are signed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQSHL variant

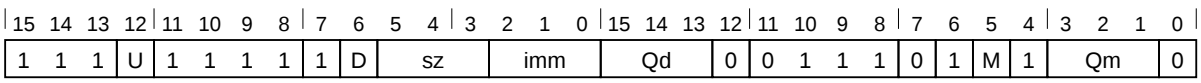
```
VQSHL<v>.<dt> Qda, Rm
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if Da == '1' then UNDEFINED;
da           = UInt(Da:Qda);
m = UInt(Rm);
d = da;
n = m;
m = da;
unsigned     = (U == '1');
destUnsigned = unsigned;
withScalar   = TRUE;
withVector   = FALSE;
esize        = 8 << UInt(size);
elements     = 32 DIV esize;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQSHL variant

```
VQSHL<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
if sz == '000' then SEE "VMOV";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d = UInt(D:Qd);
m = UInt(M:Qm);
unsigned     = (U == '1');
destUnsigned = unsigned;
n = integer UNKNOWN;
imm6         = sz:imm;
case imm6 of
    when '001xxx' size = '00'; shiftAmount = UInt(imm6) - 8;
    when '01xxxx' size = '01'; shiftAmount = UInt(imm6) - 16;
    when '1xxxxx' size = '10'; shiftAmount = UInt(imm6) - 32;
    otherwise UNDEFINED;
withScalar = FALSE;
withVector = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | 1 | D | sz | | imm | | | | Qd | | | | 0 | 0 | 1 | 1 | 0 | 0 | 1 | M | 1 | Qm | | 0 |

### T3: VQSHLU variant

```
VQSHLU<v>.<dt> Qd, Qm, #<imm>
```

## Decode for this encoding

```
if sz == '000' then SEE "VMOV";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d = UInt(D:Qd);
m = UInt(M:Qm);
unsigned     = FALSE;
destUnsigned = TRUE;
n = integer UNKNOWN;
imm6         = sz:imm;
case imm6 of
    when '001xxx' size = '00'; shiftAmount = UInt(imm6) - 8;
    when '01xxxx' size = '01'; shiftAmount = UInt(imm6) - 16;
    when '1xxxxx' size = '10'; shiftAmount = UInt(imm6) - 32;
    otherwise UNDEFINED;
withScalar = FALSE;
withVector = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 1 | 0 | D | size | | Qn | | | 0 | Qd | | | | 0 | 0 | 1 | 0 | 0 | N | 1 | M | 1 | Qm | | 0 |

### T4: VQSHL variant

```
VQSHL<v>.<dt> Qd, Qm, Qn
```

## Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d = UInt(D:Qd);
5  m = UInt(M:Qm);
6  n = UInt(N:Qn);
7  unsigned     = (U == '1');
8  destUnsigned = unsigned;
9  withScalar   = FALSE;
10 withVector   = TRUE;
11 esize        = 8 << UInt(size);
12 elements     = 32 DIV esize;
13 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | | |
|---|---|---|---|
| S8 | Encoded as | size = 00, | U = 0 |
| U8 | Encoded as | size = 00, | U = 1 |
| S16 | Encoded as | size = 01, | U = 0 |
| U16 | Encoded as | size = 01, | U = 1 |
| S32 | Encoded as | size = 10, | U = 0 |
| U32 | Encoded as | size = 10, | U = 1 |

## Assembler symbols for T2 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– The size of the elements in the vector.
This parameter must be one of the following values:

| | | | |
|---|---|---|---|
| S8 | Encoded as | sz = 001, | U = 0 |
| U8 | Encoded as | sz = 001, | U = 1 |
| S16 | Encoded as | sz = 01x, | U = 0 |
| U16 | Encoded as | sz = 01x, | U = 1 |
| S32 | Encoded as | sz = 1xx, | U = 0 |
| U32 | Encoded as | sz = 1xx, | U = 1 |

## Assembler symbols for T3 encodings

<dt> The size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | sz = 001 |
| S16 | Encoded as | sz = 01x |
| S32 | Encoded as | sz = 1xx |

## Assembler symbols for T4 encodings

<dt> This parameter determines the following values:

– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00, U = 0 |
| U8 | Encoded as | size = 00, U = 1 |
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |
| S32 | Encoded as | size = 10, U = 0 |
| U32 | Encoded as | size = 10, U = 1 |

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register, the elements of which containing the amount to shift by. |
| <Rm> | Source general-purpose register containing the amount to shift by. |
| <imm> | The number of bits to shift by, in the range 0 to <dt>-1. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
if withVector then
    op2 = Q[n, curBeat];
if withScalar then
    shiftAmount = SInt(R[n]<7:0>);
for e = 0 to elements-1
    if withVector then
        shiftAmount = SInt(Elem[op2, e, esize]<7:0>);
    operand      = Int(Elem[op1, e, esize], unsigned);
    (value, sat) = SatQ(operand << shiftAmount, esize, destUnsigned);
    Elem[result, e, esize] = value;
    if sat && elmtMask<e*(esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.429 VQSHRN

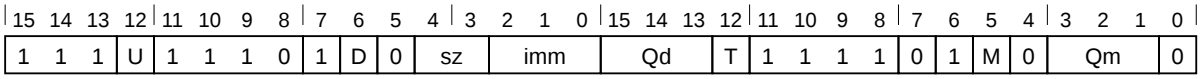
Vector Saturating Shift Right and Narrow. Performs an element-wise saturation to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQSHRN variant

```
VQSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d        = UInt(D:Qd);
4  m        = UInt(M:Qm);
5  unsigned = (U == '1');
6  imm5     = sz:imm;
7  case imm5 of
8      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
9      when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
10     otherwise UNDEFINED;
11 esize    = 8 << UInt(size);
12 elements = 16 DIV esize;
13 top      = UInt(T);
14 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– The size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as sz = 01, U = 0
- U16 Encoded as sz = 01, U = 1
- S32 Encoded as sz = 1x, U = 0
- U32 Encoded as sz = 1x, U = 1

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, 2*esize], unsigned);
    operand = operand >> shiftAmount;
    (value, sat) = SatQ(operand, esize, unsigned);
    Elem[result, 2*e + top, esize] = value;
    if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.430 VQSHRUN

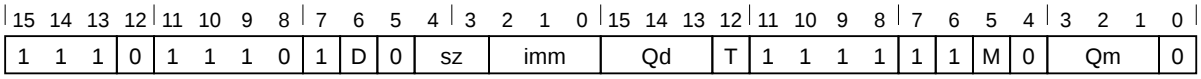
Vector Saturating Shift Right Unsigned and Narrow. Performs an element-wise saturation to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQSHRUN variant

```
VQSHRUN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d = UInt(D:Qd);
4  m = UInt(M:Qm);
5  unsigned     = FALSE;
6  destUnsigned = TRUE;
7  imm5         = sz:imm;
8  case imm5 of
9      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
10     when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
11     otherwise UNDEFINED;
12 esize    = 8 << UInt(size);
13 elements = 16 DIV esize;
14 top      = UInt(T);
15 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> The size of the elements in the vector.
This parameter must be one of the following values:
- S16 Encoded as sz = 01
- S32 Encoded as sz = 1x

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, 2*esize], unsigned);
    operand = operand >> shiftAmount;
    (value, sat) = SatQ(operand, esize, destUnsigned);
    Elem[result, 2*e + top, esize] = value;
    if sat && elmtMask<(e*2 + top) * (esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.431 VQSUB

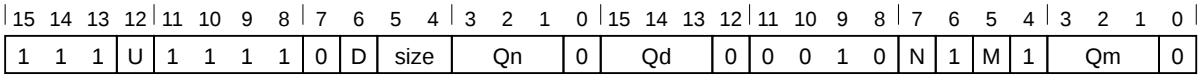
Vector Saturating Subtract. Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register. The result is saturated before being written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VQSUB variant

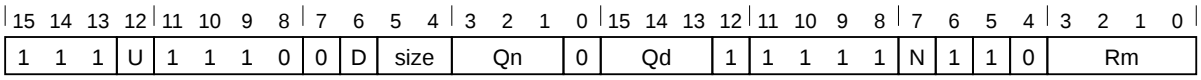
```
VQSUB<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = FALSE;
11 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VQSUB variant

```
VQSUB<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  esize      = 8 << UInt(size);
8  elements   = 32 DIV esize;
9  unsigned   = (U == '1');
10 withScalar = TRUE;
11 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
12 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Size: indicates the size of the elements in the vector.<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if !withScalar then
    op2 = Q[m, curBeat];
for e = 0 to elements-1
    if withScalar then
        value = Int(Elem[op1, e, esize], unsigned) - Int(R[m]<esize-1:0>, unsigned);
    else
        value = Int(Elem[op1, e, esize], unsigned) - Int(Elem[op2, e, esize], unsigned);
    (Elem[result, e, esize], sat) = SatQ(value, esize, unsigned);
    if sat && elmtMask<e*(esize>>3)> == '1' then
        FPSCR.QC = '1';

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.432 VREV16

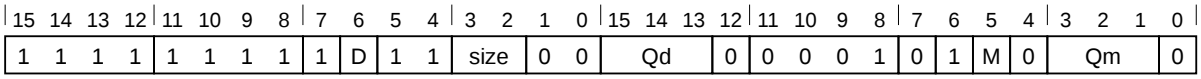
Vector Reverse. Reverse the order of elements within each halfword of the source vector register and places the result in the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VREV16 variant

`VREV16<v>.<size> Qd, Qm`

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  if size != '00'         then UNDEFINED;
4  d = UInt(D:Qd);
5  m = UInt(M:Qm);
6  esize           = 8 << UInt(size);
7  elements        = 64 DIV esize;
8  width           = 2;
9  groupsize       = (1 << (3-width-UInt(size))); // elements per reversing group: 2, 4 or 8
10 reverse_mask    = (groupsize-1)<esize-1:0>;    // EORing mask used for index calculations
11 groups_per_beat = width;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be the following value:<br>8 Encoded as `size = 00` |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  // 64 bit wide operations are handled differently as they perform cross beat
8  // register accesses
9  if width == 0 then
10     op1 = Q[m, UInt(curBeat<1:0> EOR 1<1:0>)];
11     for e = 0 to (groupsize >> 1)-1
12         // Calculate destination element index by bitwise EOR on source element index:
13         index = UInt((e<esize-1:0> EOR (UInt(reverse_mask) >> 1)<esize-1:0>));
14         Elem[result, index, esize] = Elem[op1, e, esize];
15 else
```

```
16      op1 = Q[m, curBeat];
17      for g = 0 to groups_per_beat-1
18          for e = 0 to groupsize -1
19              index = (g*groupsize) + UInt(e<esize-1:0> EOR (reverse_mask));
20              Elem[result, index, esize] = Elem[op1, (g*groupsize)+e, esize];
21
22  for e = 0 to 3
23      if elmtMask<e> == '1' then
24          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.433 VREV32

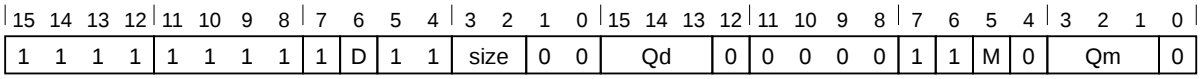
Vector Reverse. Reverse the order of elements within each word of the source vector register and places the result in the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VREV32 variant

`VREV32<v>.<size> Qd, Qm`

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '1x'            then UNDEFINED;
4  d = UInt(D:Qd);
5  m = UInt(M:Qm);
6  esize           = 8 << UInt(size);
7  elements        = 64 DIV esize;
8  width           = 1;
9  groupsize       = (1 << (3-width-UInt(size))); // elements per reversing group: 2, 4 or 8
10 reverse_mask    = (groupsize-1)<esize-1:0>;    // EORing mask used for index calculations
11 groups_per_beat = width;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as size = 00<br>16 Encoded as size = 01 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  // 64 bit wide operations are handled differently as they perform cross beat
8  // register accesses
9  if width == 0 then
10     op1 = Q[m, UInt(curBeat<1:0> EOR 1<1:0>)];
11     for e = 0 to (groupsize >> 1)-1
12         // Calculate destination element index by bitwise EOR on source element index:
13         index = UInt((e<esize-1:0> EOR (UInt(reverse_mask) >> 1)<esize-1:0>));
```

```
14          Elem[result, index, esize] = Elem[op1, e, esize];
15  else
16      op1 = Q[m, curBeat];
17      for g = 0 to groups_per_beat-1
18          for e = 0 to groupsize -1
19              index = (g*groupsize) + UInt(e<esize-1:0> EOR (reverse_mask));
20              Elem[result, index, esize] = Elem[op1, (g*groupsize)+e, esize];
21
22  for e = 0 to 3
23      if elmtMask<e> == '1' then
24          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.434 VREV64

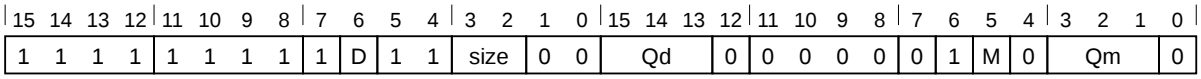
Vector Reverse. Reverse the order of elements within each doubleword of the source vector register and places the result in the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VREV64 variant

```
VREV64<v>.<size> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d = UInt(D:Qd);
5  m = UInt(M:Qm);
6  esize           = 8 << UInt(size);
7  elements        = 64 DIV esize;
8  width           = 0;
9  groupsize       = (1 << (3-width-UInt(size))); // elements per reversing group: 2, 4 or 8
10 reverse_mask    = (groupsize-1)<esize-1:0>;    // EORing mask used for index calculations
11 groups_per_beat = width;
12 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
13 if D:Qd == M:Qm then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as `size = 00`<br>16 Encoded as `size = 01`<br>32 Encoded as `size = 10` |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  result = Zeros(32);
7  // 64 bit wide operations are handled differently as they perform cross beat
8  // register accesses
9  if width == 0 then
10     op1 = Q[m, UInt(curBeat<1:0> EOR 1<1:0>)];
11     for e = 0 to (groupsize >> 1)-1
```

```
        // Calculate destination element index by bitwise EOR on source element index:
        index = UInt((e<esize-1:0> EOR (UInt(reverse_mask) >> 1)<esize-1:0>));
        Elem[result, index, esize] = Elem[op1, e, esize];
else
    op1 = Q[m, curBeat];
    for g = 0 to groups_per_beat-1
        for e = 0 to groupsize -1
            index = (g*groupsize) + UInt(e<esize-1:0> EOR (reverse_mask));
            Elem[result, index, esize] = Elem[op1, (g*groupsize)+e, esize];

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.435 VRHADD

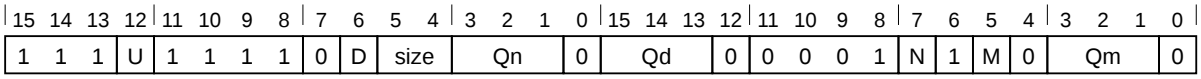
Vector Rounding Halving Add. Add the value of the elements in the first source vector register to the respective elements in the second source vector register. The result is halved and rounded before being written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRHADD variant

```
VRHADD<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  n        = UInt(N:Qn);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  unsigned = (U == '1');
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

### Operation for all encodings

```
1 EncodingSpecificOperations();
2 ExecuteFPCheck();
3
4 (curBeat, elmtMask) = GetCurInstrBeat();
5
```

```
result = Zeros(32);
op1    = Q[n, curBeat];
op2    = Q[m, curBeat];
for e = 0 to elements-1
    value = Int(Elem[op1, e, esize], unsigned) + Int(Elem[op2, e, esize], unsigned);
    Elem[result, e, esize] = (value + 1)<esize:1>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.436 VRINT (floating-point)

Vector Round Integer. Round a floating-point value to an integer value. The result remains in floating-point format. It is not converted to an integer.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | 1 | D | 1 | 1 | size | | 1 | 0 | Qd | | | | 0 | 0 | 1 | op | | 1 | M | 0 | Qm | | | 0 |

#### T1: VRINT variant

```
VRINT<op><v>.<dt> Qd, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D  == '1' || M == '1' then UNDEFINED;
3  if op == '1x0'           then UNDEFINED;
4  if size IN {'11', '00'}  then UNDEFINED;
5  d        = UInt(D:Qd);
6  m        = UInt(M:Qm);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  case op of
10     when '010' // A, Round to nearest, with ties away
11         rmode = '01'; away = TRUE;  exact = FALSE;
12     when '000' // N, Round to nearest, with ties to even
13         rmode = '00'; away = FALSE; exact = FALSE;
14     when '111' // P, Round towards Plus Infinity
15         rmode = '01'; away = FALSE; exact = FALSE;
16     when '101' // M, Round towards Minus Infinity
17         rmode = '10'; away = FALSE; exact = FALSE;
18     when '001' // X, Round to nearest with ties to even, raising inexact
19                //    exception if result not numerically equal to input
20         rmode = '00'; away = FALSE; exact = TRUE;
21     when '011' // Z, Round towards zero
22         rmode = '11'; away = FALSE; exact = FALSE;
23     otherwise
24         // Other case are UNDEFINED
25 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<op> The rounding mode.
This parameter must be one of the following values:

- N Encoded as op = 000
  Round to nearest with ties to even
- X Encoded as op = 001
  Round to nearest with ties to even, raising inexact exception if result not numerically equal to input
- A Encoded as op = 010
  Round to nearest with ties to away
- Z Encoded as op = 011
  Round towards zero

M Encoded as op = 101
Round towards minus infinity
P Encoded as op = 111
Round towards plus infinity

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Size: indicates the floating-point format used.
This parameter must be one of the following values:
F16 Encoded as size = 01
F32 Encoded as size = 10

<Qd> Destination vector register.

<Qm> Source vector register.

## Operation for all encodings

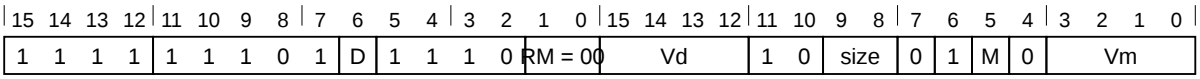
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    Elem[result, e, esize] = FPRoundInt(Elem[op1, e, esize], rmode, away, exact);

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.437 VRINTA

Floating-point Round to Nearest Integer with Ties to Away. Floating-point Round to Nearest Integer with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTA{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTA{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTA{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  case RM of
6      when '00'  // Round to nearest, with ties away
7          rmode = '01'; away = TRUE;
8      when '01'  // Round to nearest, with ties to even
9          rmode = '00'; away = FALSE;
10     when '10'  // Round towards Plus Infinity
11         rmode = '01'; away = FALSE;
12     when '11'  // Round towards Minus Infinity
13         rmode = '10'; away = FALSE;
14 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
15 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

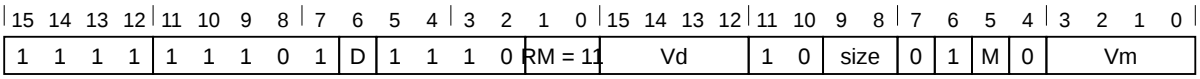
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.438 VRINTM

Floating-point Round to Integer towards -Infinity. Floating-point Round to Integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTM{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTM{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTM{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  case RM of
6      when '00'  // Round to nearest, with ties away
7          rmode = '01'; away = TRUE;
8      when '01'  // Round to nearest, with ties to even
9          rmode = '00'; away = FALSE;
10     when '10'  // Round towards Plus Infinity
11         rmode = '01'; away = FALSE;
12     when '11'  // Round towards Minus Infinity
13         rmode = '10'; away = FALSE;
14 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
15 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

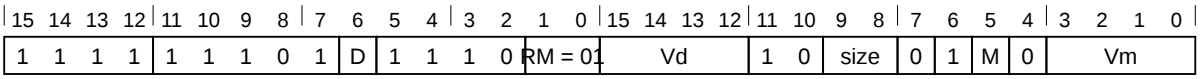
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.439 VRINTN

Floating-point Round to Nearest Integer with Ties to Even. Floating-point Round to Nearest Integer with Ties to Even rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only,* size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTN{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTN{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTN{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
case RM of
    when '00'  // Round to nearest, with ties away
        rmode = '01'; away = TRUE;
    when '01'  // Round to nearest, with ties to even
        rmode = '00'; away = FALSE;
    when '10'  // Round towards Plus Infinity
        rmode = '01'; away = FALSE;
    when '11'  // Round towards Minus Infinity
        rmode = '10'; away = FALSE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

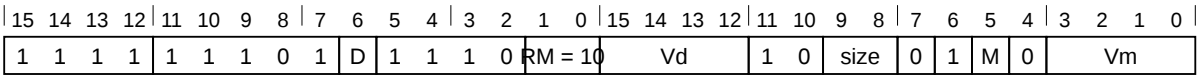
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.440 VRINTP

Floating-point Round to Integer towards +Infinity. Floating-point Round to Integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTP{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTP{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTP{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  case RM of
6      when '00'  // Round to nearest, with ties away
7          rmode = '01'; away = TRUE;
8      when '01'  // Round to nearest, with ties to even
9          rmode = '00'; away = FALSE;
10     when '10'  // Round towards Plus Infinity
11         rmode = '01'; away = FALSE;
12     when '11'  // Round towards Minus Infinity
13         rmode = '10'; away = FALSE;
14 d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
15 m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

## Operation for all encodings

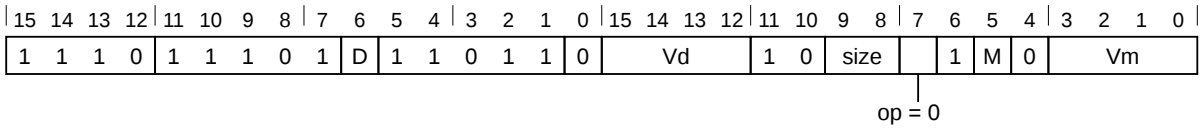
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.441 VRINTR

Floating-point Round to Integer. Floating-point Round to Integer rounds a floating-point value to an integral floating-point value of the same size using the rounding mode specified in FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTR{<c>}{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTR{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTR{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();

    rmode = if op == '1' then '11' else FPSCR.RMode;
    exact = FALSE;
    away  = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

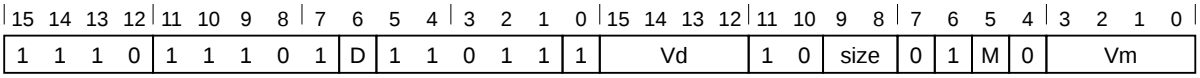
## C2.4.442 VRINTX

Floating-point Round to Integer, raising Inexact exception. This instruction rounds a floating-point value to an integral floating-point value of the same size. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

VRINTX uses the rounding mode specified in FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTX{<c>}{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTX{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTX{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
```

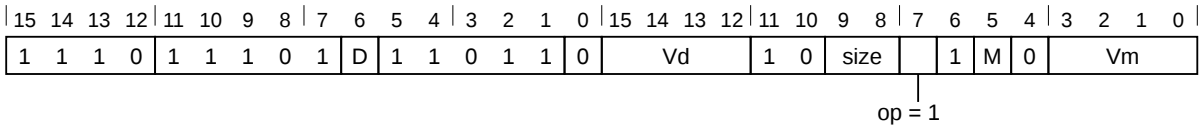
```
    EncodingSpecificOperations();
    ExecuteFPCheck();

    rmode = FPSCR<23:22>;
    away  = FALSE;
    exact = TRUE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.443 VRINTZ

Floating-point Round to Integer towards Zero. Floating-point Round to Integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VRINTZ{<c>}{<q>}.F16.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VRINTZ{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VRINTZ{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();

    rmode = if op == '1' then '11' else FPSCR.RMode;
    exact = FALSE;
    away  = FALSE;

    case size of
        when '01'
            S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, rmode, away, exact);
        when '10'
            S[d] = FPRoundInt(S[m], rmode, away, exact);
        when '11'
            D[d] = FPRoundInt(D[m], rmode, away, exact);
```

## C2.4.444 VRMLALDAVH

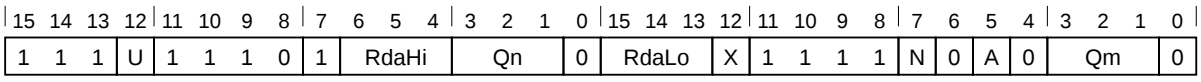
Vector Rounding Multiply Add Long Dual Accumulate Across Vector Returning High 64 bits. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated. The upper 64 bits of a 72-bit accumulator value is selected and stored across two registers, the top 32 bits are stored in an even-numbered register and the lower 32 bits are stored in an odd-numbered register. The initial value of the general-purpose destination registers can optionally be shifted up by 8 bits and added to the result. The result is rounded before the top 64 bits are selected.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRMLALDAVH variant

```
VRMLALDAVH{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

### Decode for this encoding

```
1  if RdaHi == '11x' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if N == '1'             then UNDEFINED;
4  if U == '1' && X == '1' then UNDEFINED;
5  dah         = UInt(RdaHi:'1');
6  dal         = UInt(RdaLo:'0');
7  m           = UInt(Qm);
8  n           = UInt(N:Qn);
9  exchange    = (X == '1');
10 accumulate  = (A == '1');
11 esize       = 32;
12 elements    = 32 DIV esize;
13 unsigned    = (U == '1');
14 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<A> Accumulate with existing register contents.
This parameter must be one of the following values:
- " Encoded as A = 0
- A Encoded as A = 1

<X> Exchange adjacent pairs of values in Qm.
This parameter must be one of the following values:
- " Encoded as X = 0
- X Encoded as X = 1

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> Unsigned flag: S indicates signed, U indicates unsigned.
This parameter must be one of the following values:
- S32 Encoded as U = 0
- U32 Encoded as U = 1

| | |
|---|---|
| <RdaLo> | General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register. |
| <RdaHi> | General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then Int(R[dah]:R[dal], unsigned) << 8 else 0;
if elmtMask<0> == '1' then
    if exchange then
        if curBeat<0> == '0' then
            mul = Int(Q[n, curBeat+1], unsigned) * Int(Q[m, curBeat], unsigned);
        else
            mul = Int(Q[n, curBeat-1], unsigned) * Int(Q[m, curBeat], unsigned);
    else
        mul = Int(Q[n, curBeat], unsigned) * Int(Q[m, curBeat], unsigned);
    result = result + mul + (1 << 7);
R[dah] = result<71:40>;
R[dal] = result<39:8>;
```

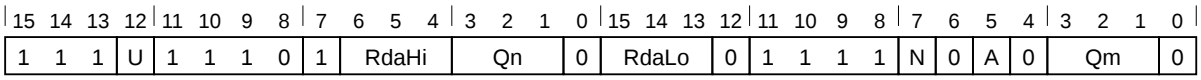
## C2.4.445 VRMLALVH

Vector Multiply Accumulate Long Across Vector Returning High 64 bits. This is an alias of VRMLALDAVH without exchange.

This is an alias of VRMLALDAVH with the following condition satisfied: X==0.

This alias is the preferred disassembly.

### T1




#### VRMLALVH variant

```
VRMLALVH{A}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

## C2.4.446 VRMLSLDAVH

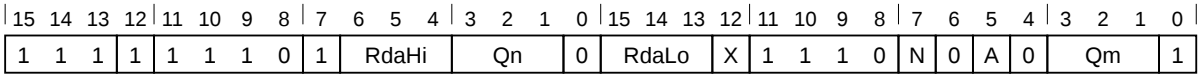
Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bits. The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated. The upper 64 bits of a 72-bit accumulator value is selected and stored across two registers, the top 32 bits are stored in an even-numbered register and the lower 32 bits are stored in an odd-numbered register. The initial value of the general-purpose destination registers can optionally be shifted up by 8 bits and added to the result. The result is rounded before the top 64 bits are selected.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRMLSLDAVH variant

```
VRMLSLDAVH{A}{X}<v>.S32 RdaLo, RdaHi, Qn, Qm
```

### Decode for this encoding

```
1  if RdaHi == '111' then SEE "VMLSDAV";
2  CheckDecodeFaults(ExtType_Mve);
3  if N == '1' then UNDEFINED;
4  dah         = UInt(RdaHi:'1');
5  dal         = UInt(RdaLo:'0');
6  m           = UInt(Qm);
7  n           = UInt(N:Qn);
8  exchange    = (X == '1');
9  accumulate  = (A == '1');
10 esize       = 32;
11 elements    = 32 DIV esize;
12 if InITBlock()    then CONSTRAINED_UNPREDICTABLE;
13 if RdaHi == '110' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<A> Accumulate with existing register contents.
This parameter must be one of the following values:
- " Encoded as A = 0
- A Encoded as A = 1

<X> Exchange adjacent pairs of values in Qm.
This parameter must be one of the following values:
- " Encoded as X = 0
- X Encoded as X = 1

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<RdaLo> General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register.

<RdaHi> General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

| | |
|---|---|
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = if accumulate || !IsFirstBeat() then SInt(R[dah]:R[dal]) << 8 else 0;
if elmtMask<0> == '1' then
    if exchange then
        if curBeat<0> == '0' then
            mul = SInt(Q[n, curBeat+1]) * SInt(Q[m, curBeat]);
        else
            mul = SInt(Q[n, curBeat-1]) * SInt(Q[m, curBeat]);
    else
        mul = SInt(Q[n, curBeat]) * SInt(Q[m, curBeat]);
    if curBeat<0> == '0' then
        result = result + mul + (1 << 7);
    else
        result = result - mul + (1 << 7);
R[dah] = result<71:40>;
R[dal] = result<39:8>;
```

## C2.4.447 VRSHL

Vector Rounding Shift Left. The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

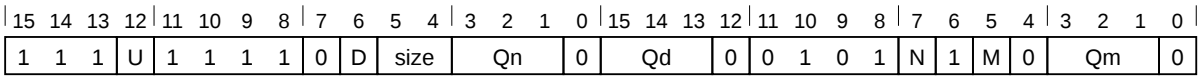
The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRSHL variant

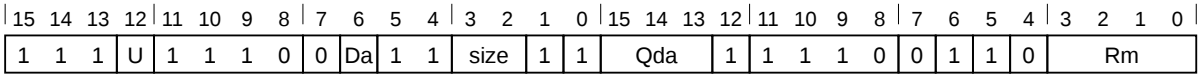
```
VRSHL<v>.<dt> Qd, Qm, Qn
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  unsigned   = (U == '1');
8  withScalar = FALSE;
9  withVector = TRUE;
10 esize      = 8 << UInt(size);
11 elements   = 32 DIV esize;
12 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VRSHL variant

```
VRSHL<v>.<dt> Qda, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if Da == '1' then UNDEFINED;
4  da         = UInt(Da:Qda);
5  m          = UInt(Rm);
6  d          = da;
7  n          = m;
8  m          = da;
9  unsigned   = (U == '1');
10 withScalar = TRUE;
```

```
withVector = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as size = 00, U = 0<br>U8 Encoded as size = 00, U = 1<br>S16 Encoded as size = 01, U = 0<br>U16 Encoded as size = 01, U = 1<br>S32 Encoded as size = 10, U = 0<br>U32 Encoded as size = 10, U = 1 |
| <Qd> | Destination vector register. |
| <Qda> | Source and destination vector register. |
| <Qm> | Source vector register. |
| <Qn> | Source vector register, the elements of which containing the amount to shift by. |
| <Rm> | Source general-purpose register containing the amount to shift by. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
if withVector then
    op2 = Q[n, curBeat];
if withScalar then
    shiftAmount = SInt(R[n]<7:0>);
for e = 0 to elements-1
    if withVector then
        shiftAmount = SInt(Elem[op2, e, esize]<7:0>);
    // 0 for left shift, 2^(n-1) for right shift
    round_const = 1 << (-1-shiftAmount);
    operand     = Int(Elem[op1, e, esize], unsigned);
    Elem[result, e, esize] = ((operand + round_const) << shiftAmount)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.448 VRSHR

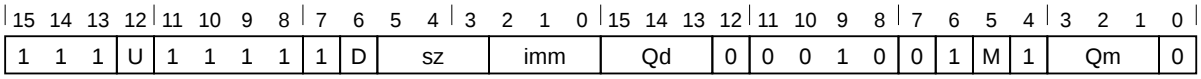
Vector Rounding Shift Right. The immediate variant shifts each element of a vector register to the right by the immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRSHR variant

```
VRSHR<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  if sz == '000' then SEE "VMOV";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' then UNDEFINED;
4  d         = UInt(D:Qd);
5  m         = UInt(M:Qm);
6  unsigned  = (U == '1');
7  n         = integer UNKNOWN;
8  imm6      = sz:imm;
9  case imm6 of
10     when '001xxx'  size = '00'; shiftAmount = 16 - UInt(imm6);
11     when '01xxxx'  size = '01'; shiftAmount = 32 - UInt(imm6);
12     when '1xxxxx' size  = '10'; shiftAmount = 64 - UInt(imm6);
13     otherwise UNDEFINED;
14 withScalar = FALSE;
15 withVector = FALSE;
16 esize      = 8 << UInt(size);
17 elements   = 32 DIV esize;
18 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– The size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as sz = 001, U = 0<br>U8 Encoded as sz = 001, U = 1<br>S16 Encoded as sz = 01x, U = 0<br>U16 Encoded as sz = 01x, U = 1<br>S32 Encoded as sz = 1xx, U = 0<br>U32 Encoded as sz = 1xx, U = 1 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |
| <imm> | The number of bits to shift by, in the range 1 to <dt>. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
if withVector then
    op2 = Q[n, curBeat];
if withScalar then
    shiftAmount = SInt(R[n]<7:0>);
for e = 0 to elements-1
    if withVector then
        shiftAmount = SInt(Elem[op2, e, esize]<7:0>);
    // 0 for left shift, 2^(n-1) for right shift
    round_const = 1 << (shiftAmount-1);
    operand     = Int(Elem[op1, e, esize], unsigned);
    Elem[result, e, esize] = ((operand + round_const) >> shiftAmount)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.449 VRSHRN

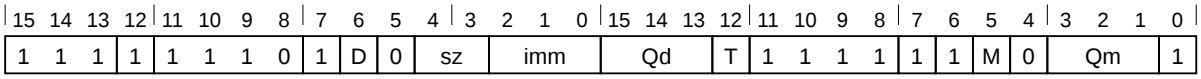
Vector Rounding Shift Right and Narrow. Performs an element-wise narrowing to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VRSHRN variant

```
VRSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d    = UInt(D:Qd);
4  m    = UInt(M:Qm);
5  imm5 = sz:imm;
6  case imm5 of
7      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
8      when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
9      otherwise UNDEFINED;
10 esize    = 8 << UInt(size);
11 elements = 16 DIV esize;
12 top      = UInt(T);
13 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> The size of the elements in the vector.
This parameter must be one of the following values:
- I16 Encoded as sz = 01
- I32 Encoded as sz = 1x

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = UInt(Elem[op1, e, 2*esize]);
    // 0 for left shift, 2^(n-1) for right shift
    operand = operand + (1 << (shiftAmount-1));
    operand = operand >> shiftAmount;
    Elem[result, 2*e + top, esize] = operand<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.450 VSBC

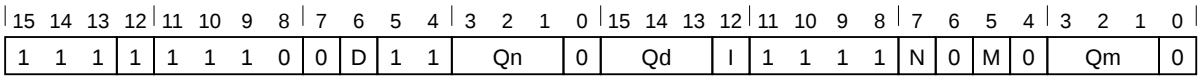
Whole Vector Subtract With Carry. Beat-wise subtracts the value of the elements in the second source vector register and the value of NOT(Carry flag) from the respective elements in the first source vector register, the carry flag being FPSCR.C. The initial value of FPSCR.C can be overridden by using the I variant. FPSCR.C is not updated for beats disabled due to predication. FPSCR.N, .V and .Z are zeroed.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSBC variant

```
VSBC{I}<v>.I32 Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d         = UInt(D:Qd);
4  m         = UInt(M:Qm);
5  n         = UInt(N:Qn);
6  carryInit = (I == '1');
7  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<I> Specifies where the initial carry in for wide arithmetic comes from.
This parameter must be one of the following values:
- ” Encoded as `I = 0`
  Indicates carry input comes from FPSCR.C.
- `I` Encoded as `I = 1`
  Indicates carry input is 1.

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Qd> Destination vector register.
<Qn> First source vector register.
<Qm> Second source vector register.

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  op1 = Q[n, curBeat];
7  op2 = NOT(Q[m, curBeat]);
8  if carryInit && IsFirstBeat() then
9      (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = ('0', '0', '1', '0');
10 (result, carryOut, -) = AddWithCarry(op1, op2, FPSCR.C);
11 if elmtMask<0> == '1' then
12     (FPSCR.N, FPSCR.Z, FPSCR.C, FPSCR.V) = ('0', '0', carryOut, '0');
```

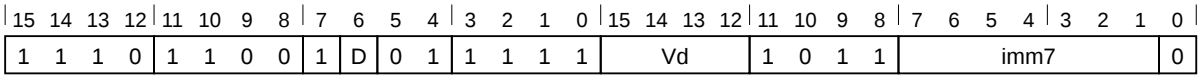
```
13
14  for e = 0 to 3
15      if elmtMask<e> == '1' then
16          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.451 VSCCLRM

Floating-point Secure Context Clear Multiple. Zeros VPR and the specified floating-point registers if there is an active floating-point context. This instruction is UNDEFINED if executed in Non-secure state. This instruction is present on all PEs that implement the Armv8.1-M architecture, even if the Floating-point Extension is not present. It is IMPLEMENTATION DEFINED whether this instruction is exception-continuable. See EPSR.ICI. If an exception returns to this instruction with non-zero EPSR.ICI bits, and the PE does not support exception-continuable behavior, the instruction restarts from the beginning. If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0

### T1

*Armv8.1-M Main Extension only*




#### T1: VSCCLRM variant

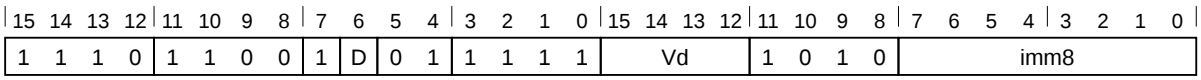
```
VSCCLRM<c> <dreglist>
```

### Decode for this encoding

```
// If the Armv8.1-M extensions aren't implemented this instruction will raise a
// NOCP or an UNDEFINSTR UsageFault depending on whether the co-processor is
// enabled.
if !HasArchVersion(Armv8p1) then
    HandleException(CheckCPEnabled(10));
    UNDEFINED;
if !HaveMainExt() || !IsSecure() then
    UNDEFINED;
if HaveFPExt() && (FPCCR.ASPEN == '0' || CONTROL_S.SFPA == '1') then
    HandleException(CheckCPEnabled(10));
singleRegs = FALSE;
d          = UInt(D:Vd);
regs       = UInt(imm7);
topReg     = d+regs-1;
if topReg > 31 then UNPREDICTABLE;
```

### T2

*Armv8.1-M Main Extension only*




#### T2: VSCCLRM variant

```
VSCCLRM<c> <sreglist>
```

### Decode for this encoding

```
// If the Armv8.1-M extensions aren't implemented this instruction will raise a
// NOCP or an UNDEFINSTR UsageFault depending on whether the co-processor is
// enabled.
if !HasArchVersion(Armv8p1) then
    HandleException(CheckCPEnabled(10));
```

```
    UNDEFINED;
if !HaveMainExt() || !IsSecure() then
    UNDEFINED;
if HaveFPExt() && (FPCCR.ASPEN == '0' || CONTROL_S.SFPA == '1') then
    HandleException(CheckCPEnabled(10));
singleRegs = TRUE;
d          = UInt(Vd:D);
regs       = UInt(imm8);
topReg     = d+regs-1;
if topReg > 63                    then UNPREDICTABLE;
if topReg > 31 && topReg<0> == '0' then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be cleared. The first register in the list is encoded in "D:Vd", and "imm7" is set to the number of registers in the list. Because this instruction always clears the VPR register, it is mandatory to have VPR in the register list |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be cleared. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. Registers above S31 are specified by using D registers in the register list. Because this instruction always clears the VPR register, it is mandatory to have VPR in the register list |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    if HaveFPExt() && (FPCCR.ASPEN == '0' || CONTROL_S.SFPA == '1') then
        ExecuteFPCheck();

        for r = 0 to regs-1
            if singleRegs then
                if (d+r) < 32 || !VFPSmallRegisterBank() then
                    S[d+r] = Zeros();
            else
                if (d+r) < 16 || !VFPSmallRegisterBank() then
                    D[d+r] = Zeros();
        VPR = Zeros(32);
```

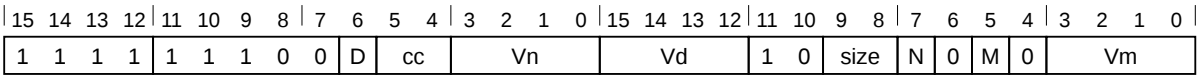
## C2.4.452 VSEL

Floating-point Conditional Select. Floating-point Conditional Select allows the destination register to take the value from either one or the other of two source registers according to the condition codes in the APSR.

The condition codes for VSEL are limited to GE, GT, EQ, and VS. The effect of LT, LE, NE, and VC can be achieved by exchanging the source operands.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**VSELEQ, Double-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 00 && size == 11`.**

```
VSELEQ.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

**VSELEQ, Half-precision variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `cc == 00 && size == 01`.**

```
VSELEQ.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELEQ, Single-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 00 && size == 10`.**

```
VSELEQ.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELGE, Double-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 10 && size == 11`.**

```
VSELGE.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

**VSELGE, Half-precision variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `cc == 10 && size == 01`.**

```
VSELGE.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELGE, Single-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 10 && size == 10`.**

```
VSELGE.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELGT, Double-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 11 && size == 11`.**

```
VSELGT.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

**VSELGT, Half-precision variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `cc == 11 && size == 01`.**

```
VSELGT.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELGT, Single-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 11 && size == 10`.**

```
VSELGT.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELVS, Double-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 01 && size == 11`.**

```
VSELVS.F64 <Dd>, <Dn>, <Dm>
  // Not permitted in IT block
```

**VSELVS, Half-precision variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `cc == 01 && size == 01`.**

```
VSELVS.F16 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

**VSELVS, Single-precision variant**

***Armv8-M Floating-point Extension only.***

**Applies when `cc == 01 && size == 10`.**

```
VSELVS.F32 <Sd>, <Sn>, <Sm>
  // Not permitted in IT block
```

## Decode for this encoding

```
dp_operation = (size == '11');
CheckFPDecodeFaults(size);
if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
    UNDEFINED;
if InITBlock() then UNPREDICTABLE;
cond = cc:(cc<1> EOR cc<0>):'0';
d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

## Assembler symbols for all encodings

| | |
|---|---|
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

case size of
    when '01'
        S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
    when '10'
        S[d] = if ConditionHolds(cond) then S[n] else S[m];
    when '11'
        D[d] = if ConditionHolds(cond) then D[n] else D[m];
```

## C2.4.453 VSHL

Vector Shift Left. The immediate variant shifts each element of a vector register to the left by the immediate value.

The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | 1 | 1 | D | sz | | imm | | | | Qd | | | | 0 | 0 | 1 | 0 | 1 | 0 | 1 | M | 1 | Qm | | 0 |

#### T1: VSHL variant

```
VSHL<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
if sz == '000' then SEE "VMOV";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d        = UInt(D:Qd);
m        = UInt(M:Qm);
unsigned = TRUE;
n        = integer UNKNOWN;
imm6     = sz:imm;
case imm6 of
    when '001xxx' size = '00'; shiftAmount = UInt(imm6) - 8;
    when '01xxxx' size = '01'; shiftAmount = UInt(imm6) - 16;
    when '1xxxxx' size = '10'; shiftAmount = UInt(imm6) - 32;
    otherwise UNDEFINED;
withScalar = FALSE;
withVector = FALSE;
esize      = 8 << UInt(size);
elements   = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 0 | 0 | Da | 1 | 1 | size | | 0 | 1 | Qda | | | | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | Rm | | |

#### T2: VSHL variant

```
VSHL<v>.<dt> Qda, Rm
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if Da == '1' then UNDEFINED;
da          = UInt(Da:Qda);
m           = UInt(Rm);
d           = da;
n           = m;
m           = da;
unsigned    = (U == '1');
withScalar = TRUE;
withVector = FALSE;
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 1 | 0 | D | size | | Qn | | | 0 | Qd | | | 0 | 0 | 1 | 0 | 0 | N | 1 | M | 0 | Qm | | | 0 |

### T3: VSHL variant

`VSHL<v>.<dt> Qd, Qm, Qn`

## Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' || N == '1' then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(N:Qn);
unsigned    = (U == '1');
withScalar = FALSE;
withVector = TRUE;
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

`<dt>` The size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| `I8` | Encoded as | `sz = 001` |
| `I16` | Encoded as | `sz = 01x` |
| `I32` | Encoded as | `sz = 1xx` |

## Assembler symbols for T2 encodings

`<dt>` This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| `S8` | Encoded as | `size = 00, U = 0` |
| `U8` | Encoded as | `size = 00, U = 1` |
| `S16` | Encoded as | `size = 01, U = 0` |

```
U16  Encoded as  size = 01,  U = 1
S32  Encoded as  size = 10,  U = 0
U32  Encoded as  size = 10,  U = 1
```

### Assembler symbols for T3 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

```
S8   Encoded as  size = 00,  U = 0
U8   Encoded as  size = 00,  U = 1
S16  Encoded as  size = 01,  U = 0
U16  Encoded as  size = 01,  U = 1
S32  Encoded as  size = 10,  U = 0
U32  Encoded as  size = 10,  U = 1
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Qd> Destination vector register.
<Qda> Source and destination vector register.
<Qm> Source vector register.
<Qn> Source vector register, the elements of which containing the amount to shift by.
<Rm> Source general-purpose register containing the amount to shift by.
<imm> The number of bits to shift by, in the range 0 to <dt>-1. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
if withVector then
    op2 = Q[n, curBeat];
if withScalar then
    shiftAmount = SInt(R[n]<7:0>);
for e = 0 to elements-1
    if withVector then
        shiftAmount = SInt(Elem[op2, e, esize]<7:0>);
    operand = Int(Elem[op1, e, esize], unsigned);
    Elem[result, e, esize] = (operand << shiftAmount)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.454 VSHLC

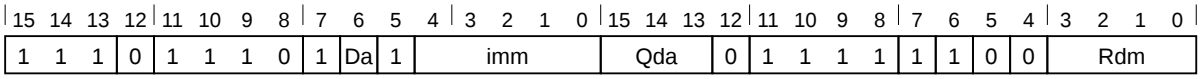
Whole Vector Left Shift with Carry. Logical shift left by 1-32 bits, with carry across beats, carry in from general-purpose register, and carry out to the same general-purpose register. Permits treating a vector register as a single 128-bit scalar. The carry in is from the lower <imm> bits of the general-purpose register, not the upper bits.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSHLC variant

```
VSHLC<v> Qda, Rdm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if Da == '1' then UNDEFINED;
3  da = UInt(Da:Qda);
4  dm = UInt(Rdm);
5  (-, amount) = DecodeImmShift('10', imm);
6  if InITBlock()   then CONSTRAINED_UNPREDICTABLE;
7  if Rdm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qda> | Source and destination vector register. |
| <Rdm> | Source and destination general-purpose register for carry in and out. |
| <imm> | The number of bits to shift by, in the range 1-32. |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
6  (result, carry) = LSL_C(Q[da, curBeat], R[dm], amount);
7  if elmtMask<0> == '1' then
8      R[dm] = carry;
9
10 for e = 0 to 3
11     if elmtMask<e> == '1' then
12         Elem[Q[da, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.455 VSHLL

Vector Shift Left Long. Selects an element of 8 or 16-bits from either the top half (T variant) or bottom half (B variant) of each source element, performs a left shift by an immediate value, performs a signed or unsigned left shift by an immediate value and places the 16 or 32-bit results in the destination vector.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 0 | 1 | D | 1 | sz | | | imm | | | | Qd | | T | 1 | 1 | 1 | 1 | 0 | 1 | M | 0 | | Qm | | 0 |

#### T1: VSHLL variant

```
VSHLL<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
if imm == '000' && sz IN {'10', '01'} then SEE "VMOVL";
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d        = UInt(D:Qd);
m        = UInt(M:Qm);
unsigned = (U == '1');
imm5     = sz:imm;
case imm5 of
    when '01xxx' size = '00'; shiftAmount = UInt(imm5) - 8;
    when '1xxxx' size = '01'; shiftAmount = UInt(imm5) - 16;
    otherwise UNDEFINED;
esize    = 8 << UInt(size);
elements = 16 DIV esize;
top      = UInt(T);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | U | 1 | 1 | 1 | 0 | 0 | D | 1 | 1 | size | | 0 | 1 | | | Qd | | T | 1 | 1 | 1 | 0 | 0 | 0 | M | 0 | | Qm | | 1 |

#### T2: VSHLL variant

```
VSHLL<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D    == '1' || M == '1' then UNDEFINED;
if size == '10'            then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
unsigned    = (U == '1');
esize       = 8 << UInt(size);
elements    = 16 DIV esize;
```

```
shiftAmount = esize;
top         = UInt(T);
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for T1 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– The size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | sz = 01, U = 0 |
| U8 | Encoded as | sz = 01, U = 1 |
| S16 | Encoded as | sz = 1x, U = 0 |
| U16 | Encoded as | sz = 1x, U = 1 |

### Assembler symbols for T2 encodings

<dt> This parameter determines the following values:
– Unsigned flag: S indicates signed, U indicates unsigned.
– Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:

| | | |
|---|---|---|
| S8 | Encoded as | size = 00, U = 0 |
| U8 | Encoded as | size = 00, U = 1 |
| S16 | Encoded as | size = 01, U = 0 |
| U16 | Encoded as | size = 01, U = 1 |

### Assembler symbols for all encodings

<T> Specifies which half of the source element is used.
This parameter must be one of the following values:

| | | |
|---|---|---|
| B | Encoded as | T = 0 |
| | Indicates bottom half | |
| T | Encoded as | T = 1 |
| | Indicates top half | |

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Qd> Destination vector register.
<Qm> Source vector register.
<imm> The number of bits to shift by, in the range 1 to <dt>. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). If <imm> == <dt> the encoding is T2, otherwise the encoding is T1

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, 2*e + top, esize], unsigned);
    operand = operand << shiftAmount;
    Elem[result, e, 2*esize] = operand<(2*esize)-1:0>;

```

```
13  for e = 0 to 3
14      if elmtMask<e> == '1' then
15          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.456 VSHR

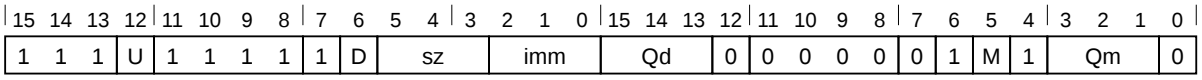
Vector Shift Right. Shifts each element of a vector register to the right by the immediate value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSHR variant

```
VSHR<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  if sz == '000' then SEE "VMOV";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  unsigned = (U == '1');
7  imm6     = sz:imm;
8  case imm6 of
9      when '001xxx'  size = '00'; shiftAmount = 16 - UInt(imm6);
10     when '01xxxx'  size = '01'; shiftAmount = 32 - UInt(imm6);
11     when '1xxxxx' size  = '10'; shiftAmount = 64 - UInt(imm6);
12     otherwise UNDEFINED;
13 esize    = 8 << UInt(size);
14 elements = 32 DIV esize;
15 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | This parameter determines the following values:<br>– Unsigned flag: S indicates signed, U indicates unsigned.<br>– The size of the elements in the vector.<br>This parameter must be one of the following values:<br>S8 Encoded as sz = 001, U = 0<br>U8 Encoded as sz = 001, U = 1<br>S16 Encoded as sz = 01x, U = 0<br>U16 Encoded as sz = 01x, U = 1<br>S32 Encoded as sz = 1xx, U = 0<br>U32 Encoded as sz = 1xx, U = 1 |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |
| <imm> | The number of bits to shift by, in the range 1 to <dt>. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). |

### Operation for all encodings

```
1  EncodingSpecificOperations();
```

```
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[m, curBeat];
for e = 0 to elements-1
    operand = Int(Elem[op1, e, esize], unsigned);
    Elem[result, e, esize] = (operand >> shiftAmount)<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.457 VSHRN

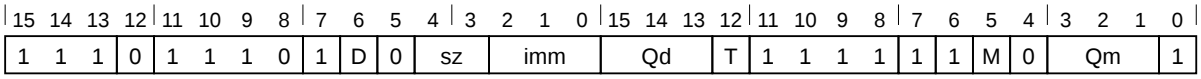
Vector Shift Right and Narrow. Performs an element-wise narrowing to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSHRN variant

```
VSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d    = UInt(D:Qd);
4  m    = UInt(M:Qm);
5  imm5 = sz:imm;
6  case imm5 of
7      when '01xxx'  size = '00'; shiftAmount = 16 - UInt(imm5);
8      when '1xxxx'  size = '01'; shiftAmount = 32 - UInt(imm5);
9      otherwise UNDEFINED;
10 esize    = 8 << UInt(size);
11 elements = 16 DIV esize;
12 top      = UInt(T);
13 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<T> Specifies which half of the result element the result is written to.
This parameter must be one of the following values:
- B Encoded as T = 0
  Indicates bottom half
- T Encoded as T = 1
  Indicates top half

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

<dt> The size of the elements in the vector.
This parameter must be one of the following values:
- I16 Encoded as sz = 01
- I32 Encoded as sz = 1x

<Qd> Destination vector register.

<Qm> Source vector register.

<imm> The number of bits to shift by, in the range 1 to <dt>/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

op1    = Q[m, curBeat];
result = Q[d, curBeat];
for e = 0 to elements-1
    operand = UInt(Elem[op1, e, 2*esize]);
    operand = operand >> shiftAmount;
    Elem[result, 2*e + top, esize] = operand<esize-1:0>;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.458 VSLI

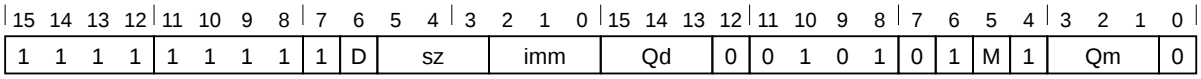
Vector Shift Left and Insert. Takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSLI variant

```
VSLI<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  if sz == '000' then SEE "VMOV";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  unsigned = TRUE;
7  imm6     = sz:imm;
8  case imm6 of
9      when '001xxx' size = '00'; shiftAmount = UInt(imm6) - 8;
10     when '01xxxx' size = '01'; shiftAmount = UInt(imm6) - 16;
11     when '1xxxxx' size = '10'; shiftAmount = UInt(imm6) - 32;
12     otherwise UNDEFINED;
13 esize    = 8 << UInt(size);
14 elements = 32 DIV esize;
15 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | The size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as sz = 001<br>16 Encoded as sz = 01x<br>32 Encoded as sz = 1xx |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |
| <imm> | The number of bits to shift by, in the range 0 to <dt>-1. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
```

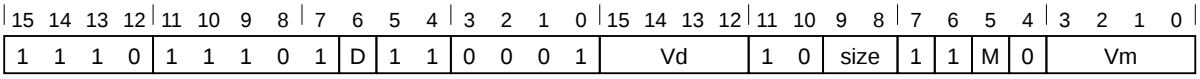
```
op1    = Q[m, curBeat];
result = Q[d, curBeat];
mask   = LSL(Ones(esize), shiftAmount);
for e = 0 to elements-1
    shiftedOp              = (LSL(Elem[op1, e, esize], shiftAmount))<esize-1:0>;
    Elem[result, e, esize] = (Elem[result, e, esize] AND NOT(mask)) OR shiftedOp;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.459 VSQRT

Floating-point Square Root. Floating-point Square Root calculates the square root of a floating-point register value and writes the result to another floating-point register.

### T1

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || D == '1') then UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sm> | Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dm> | Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteFPCheck();
4      case size of
5          when '01' S[d] = Zeros(16) : FPSqrt(S[m]<15:0>);
6          when '10' S[d] = FPSqrt(S[m]);
```

```
7           when '11' D[d] = FPSqrt(D[m]);
```

## C2.4.460 VSRI

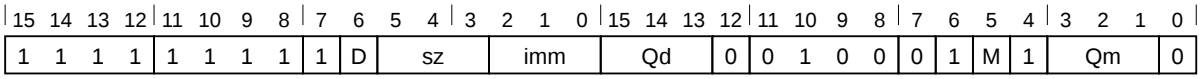
Vector Shift Right and Insert. Takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSRI variant

```
VSRI<v>.<dt> Qd, Qm, #<imm>
```

### Decode for this encoding

```
1  if sz == '000' then SEE "VMOV";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || M == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  m        = UInt(M:Qm);
6  unsigned = TRUE;
7  imm6     = sz:imm;
8  case imm6 of
9      when '001xxx'  size = '00'; shiftAmount = 16 - UInt(imm6);
10     when '01xxxx'  size = '01'; shiftAmount = 32 - UInt(imm6);
11     when '1xxxxx' size  = '10'; shiftAmount = 64 - UInt(imm6);
12     otherwise UNDEFINED;
13 esize    = 8 << UInt(size);
14 elements = 32 DIV esize;
15 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | The size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as sz = 001<br>16 Encoded as sz = 01x<br>32 Encoded as sz = 1xx |
| <Qd> | Destination vector register. |
| <Qm> | Source vector register. |
| <imm> | The number of bits to shift by, in the range 1 to <dt>. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz). |

### Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3
4  (curBeat, elmtMask) = GetCurInstrBeat();
5
```

```
op1    = Q[m, curBeat];
result = Q[d, curBeat];
mask   = LSR(Ones(esize), shiftAmount);
for e = 0 to elements-1
    shiftedOp              = (LSR(Elem[op1, e, esize], shiftAmount))<esize-1:0>;
    Elem[result, e, esize] = (Elem[result, e, esize] AND NOT(mask)) OR shiftedOp;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.461 VST2

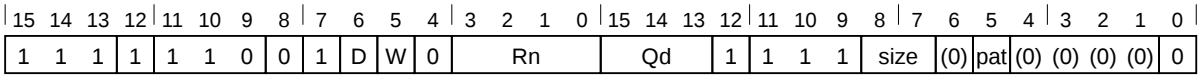
Vector Interleaving Store - Stride 2. Saves two 64-bit contiguous blocks of data to memory made up of multiple parts of 2 source registers. The parts of the source registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 2 times with the same base address and source registers, but with different pat values, the effect is to interleave the specified registers with a stride of 2 and to save the data to memory. The base address register can optionally be incremented by 32.

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VST2 variant (Non writeback: W=0)

```
VST2<pat>.<size> {Qd, Qd+1}, [Rn]
```

#### T1: VST2 variant (Writeback: W=1)

```
VST2<pat>.<size> {Qd, Qd+1}, [Rn]!
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  pattern  = UInt(pat);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  wback    = (W == '1');
10 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
11 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
12 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
13 if UInt(D:Qd) > 6           then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <pat> | Specifies the pattern of register elements and memory addresses to access.<br>This parameter must be one of the following values:<br>0 Encoded as pat = 0<br>1 Encoded as pat = 1 |
| <size> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>8 Encoded as size = 00<br>16 Encoded as size = 01<br>32 Encoded as size = 10 |
| <Qd> | Source vector register. |
| <Rn> | The base register for the target address. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, -) = GetCurInstrBeat();

// Pre-calculate variables for memory / register access patterns
addrWordOffset = curBeat<1> : (UInt(curBeat<1>) + pattern)<0> : curBeat<0>;
baseAddress    = R[n] + ZeroExtend(addrWordOffset:'00', 32);
xBeat          = UInt(curBeat<1> : (pattern<0> EOR curBeat<1>));

for e = 0 to elements-1
    address = baseAddress + (e * (esize DIV 8));
    case esize of
        when 8
            y  = UInt(e<0>);
            xE = UInt(curBeat<0> : e<1>);
        when 16
            y  = UInt(e<0>);
            xE = UInt(curBeat<0>);
        when 32
            y  = UInt(curBeat<0>);
            xE = 0;
    MemA_MVE[address, esize DIV 8] = Elem[Q[d + y, xBeat], xE, esize];

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = R[n] + 32;
```

## C2.4.462 VST4

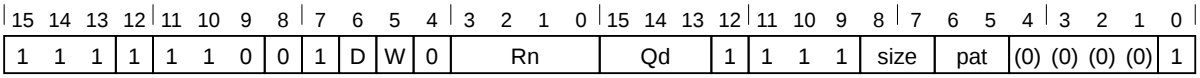
Vector Interleaving Store - Stride 4. Saves two 64-bit contiguous blocks of data to memory made up of multiple parts of 4 source registers. The parts of the source registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 4 times with the same base address and source registers, but with different pat values, the effect is to interleave the specified registers with a stride of 4 and to save the data to memory. The base address register can optionally be incremented by 64.

This instruction is not VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VST4 variant (Non writeback: W=0)

```
VST4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]
```

#### T1: VST4 variant (Writeback: W=1)

```
VST4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]!
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  pattern  = UInt(pat);
7  esize    = 8 << UInt(size);
8  elements = 32 DIV esize;
9  wback    = (W == '1');
10 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
11 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
12 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
13 if UInt(D:Qd) > 4           then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<pat> Specifies the pattern of register elements and memory addresses to access.
This parameter must be one of the following values:
- 0 Encoded as pat = 00
- 1 Encoded as pat = 01
- 2 Encoded as pat = 10
- 3 Encoded as pat = 11

<size> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- 8 Encoded as size = 00
- 16 Encoded as size = 01
- 32 Encoded as size = 10

<Qd> Source vector register.

<Rn> The base register for the target address.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, -) = GetCurInstrBeat();

// Pre-calculate variables for memory / register access patterns
addrWordOffset = curBeat<1> : (UInt(curBeat<1>) + pattern)<1:0> : curBeat<0>;
baseAddress    = R[n] + ZeroExtend(addrWordOffset:'00', 32);
xBeat          = UInt(curBeat<1> : (pattern<1> EOR (pattern<0> AND curBeat<1>)));

for e = 0 to elements-1
    address = baseAddress + (e * (esize DIV 8));
    case esize of
        when 8
            y  = UInt(e<1:0>);
            xE = UInt((pattern<0> EOR curBeat<1>) : curBeat<0>);
        when 16
            y  = UInt(curBeat<0> : e<0>);
            xE = UInt(pattern<0> EOR curBeat<1>);
        when 32
            y  = UInt((pattern<0> EOR curBeat<1>) : curBeat<0>);
            xE = 0;
    MemA_MVE[address, esize DIV 8] = Elem[Q[d + y, xBeat], xE, esize];

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = R[n] + 64;
```

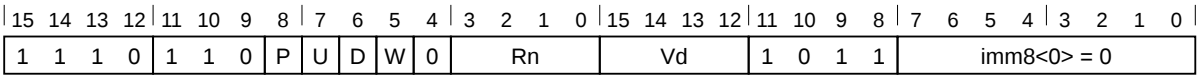
## C2.4.463 VSTM

Floating-point Store Multiple. Floating-point Store Multiple stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

This instruction is used by the alias VPUSH.

### T1

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1`.**

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

### Decode for this encoding

```
1  if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
2  if P == '1' && W == '0' then SEE VSTR;
3  CheckDecodeFaults(ExtType_MveOrFp);
4  if P == U && W == '1' then UNDEFINED;
5  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6  single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
7  d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
8  regs = UInt(imm8) DIV 2;
9  if n == 15 then UNPREDICTABLE;
10 if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

#### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

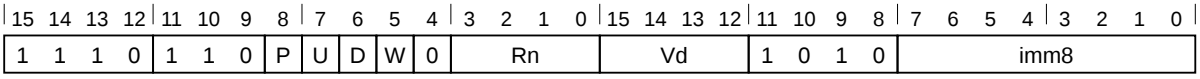
#### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 32`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

### T2

*Armv8-M Floating-point Extension only or MVE*




**Decrement Before variant**

**Applies when `P == 1 && U == 0 && W == 1`.**

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

**Increment After variant**

**Applies when `P == 0 && U == 1`.**

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

## Decode for this encoding

```
1  if P == '0' && U == '0' then SEE "Related encodings";
2  if P == '1' && W == '0' then SEE VSTR;
3  CheckDecodeFaults(ExtType_MveOrFp);
4  if P == '1' && U == '1' && W == '1' then UNDEFINED;
5  // Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
6  single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
7  imm32  = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
8  topReg = d+regs-1;
9  if n == 15                          then UNPREDICTABLE;
10 if regs == 0        || topReg > 63 then UNPREDICTABLE;
11 if topReg<0> == '0' && topReg > 31 then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a VSTM with the same addressing mode but stores no registers.

### CONSTRAINED UNPREDICTABLE behavior

If `(d+regs) > 64`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

## Alias conditions

| Alias | is preferred when |
|---|---|
| VPUSH | `P == '1' && U == '0' && W == '1' && RN == '1101'` |

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <size> | An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| ! | Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0. |
| <sreglist> | Is the list of consecutively numbered 32-bit floating-point registers to be transferred. The first register in the list is encoded in "Vd:D", and "imm8" is set to the number of registers in the list. The list must contain at least one register. |
| <dreglist> | Is the list of consecutively numbered 64-bit floating-point registers to be transferred. The first register in the list is encoded in "D:Vd", and "imm8" is set to twice the number of registers in the list. The list must contain at least one register, and must not contain more than 16 registers. |

## Operation for all encodings

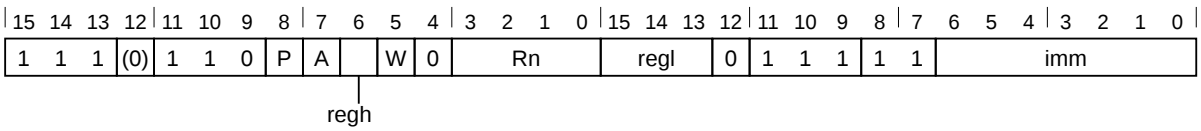
```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    address = if add then R[n]       else R[n]-imm32;
    regval  = if add then R[n]+imm32 else R[n]-imm32;

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), regval);
    else
        violatesLimit = FALSE;

    // Memory operation only performed if limit not violated
    if !violatesLimit then
        for r = 0 to regs-1
            if single_regs then
                MemA[address,4] = S[d+r];
                address         = address+4;
            else
                // Store as two word-aligned words in the correct order for current
                    endianness.
                if (d+r) < 16 || !VFPSmallRegisterBank() then
                    bigEndian         = BigEndian(address, 8);
                    MemA[address,4]   = if bigEndian then D[d+r]<63:32> else D[d+r]<31:0>;
                    MemA[address+4,4] = if bigEndian then D[d+r]<31:0>  else D[d+r]<63:32>;
                elsif boolean UNKNOWN then
                    MemA[address,4]   = bits(32) UNKNOWN;
                    MemA[address+4,4] = bits(32) UNKNOWN;
                address = address+8;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

## C2.4.464 VSTR (System Register)

Store System Register. Store a system register in memory. The target address is calculated from a base register plus an immediate offset. If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0. This instruction is UNDEFINED if executed in Non-secure state

### T1

*Armv8.1-M Floating-point Extension and / or Armv8.1-M MVE*




#### T1: VSTR variant (Offset: P=1, W=0)

```
VSTR<c> <reg>, [Rn{, #+/-<imm>}]
```

#### T1: VSTR variant (Pre-indexed: P=1, W=1)

```
VSTR<c> <reg>, [Rn, #+/-<imm>]!
```

#### T1: VSTR variant (Post-indexed: P=0, W=1)

```
VSTR<c> <reg>, [Rn], #+/-<imm>
```

### Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  if !HasArchVersion(Armv8p1) then
3      UFSR.NOCP = '1';
4      HandleException(CreateException(UsageFault));
5  fpCxtAccess    = (regh:regl == '111x');
6  fpCxtNSSAccess = (regh:regl == '1110');
7  fpInactive     = !HaveMveOrFPExt() || (FPCCR_NS.ASPEN == '1' && CONTROL.FPCA == '0');
8  if fpCxtAccess then
9      if !HaveMainExt() || !IsSecure() then UNDEFINED;
10     if !fpInactive then
11         HandleException(CheckCPEnabled(10));
12 else
13     CheckDecodeFaults(ExtType_MveOrFp);
14 n     = UInt(Rn);
15 index = (P == '1');
16 add   = (A == '1');
17 wback = (W == '1');
18 r     = regh:regl;
19 imm32 = ZeroExtend(imm:'00', 32);
20 if (regh:regl) IN {'10xx', '01xx', '0011', '0000'} then CONSTRAINED_UNPREDICTABLE;
21 if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <reg> | The system register to access<br>This parameter must be one of the following values:<br>FPSCR Encoded as regh = 0, regl = 001<br>FPSCR_nzcvqc Encoded as regh = 0, regl = 010<br>VPR Encoded as regh = 1, regl = 100<br>P0 Encoded as regh = 1, regl = 101 |

FPCXTNS Encoded as regh = 1, regl = 110
FPCXTS Encoded as regh = 1, regl = 111

<Rn> The base register for the target address.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if !fpCxtNSSAccess then
        ExecuteFPCheck();
    elsif !fpInactive then
        PreserveFPState();
    SerializeVFP();
    VFPExcBarrier();

    offsetAddr = if add   then (R[n] + imm32) else (R[n] - imm32);
    address    = if index then offsetAddr     else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        violatesLimit = ViolatesSPLim(LookUpSP(), offsetAddr);
    else
        violatesLimit = FALSE;
    // Memory operation only performed if limit not violated
    if !violatesLimit then
        case r of
            when '0001'
                MemA[address, 4] = FPSCR;
            when '0010'
                // Only read the N, Z, C, V, and QC flags
                MemA[address, 4] = FPSCR<31:27>:Zeros(27);
            when '1100'
                if HaveMve() then
                    if CurrentModeIsPrivileged() then
                        MemA[address, 4] = VPR;
                else
                    UNPREDICTABLE;
            when '1101'
                if HaveMve() then
                    MemA[address, 4] = Zeros(16):VPR.P0;
                else
                    UNPREDICTABLE;
            when '1110'
                if HaveFPExt() || HaveMve() then
                    FPCXT_Type cxt = Zeros(32);
                    if !fpInactive then
                        cxt.SFPA  = CONTROL_S.SFPA;
                        cxt<27:0> = FPSCR<27:0>;
                        // If the FP context isn't secure the FPSCR value is set
                        // to the NS default so any NS functions that are called
                        // before an FP instruction is executed in the secure
                        // state will get the same FPSCR value as functions
                        // called after a secure FP instruction (I.E. the value
                        // of FPDSCR_NS).
                        if CONTROL_S.SFPA == '0' then
                            FPSCR = FPDSCR_NS<31:0>;
                    else
                        cxt<27:0> = FPDSCR_NS<27:0>;
                    MemA[address, 4] = cxt;
            when '1111'
                FPCXT_Type cxt    = Zeros(32);
                cxt.SFPA          = CONTROL_S.SFPA;
                cxt<27:0>         = FPSCR<27:0>;
                FPSCR             = FPDSCR_NS<31:0>;
                CONTROL_S.SFPA    = '0';
```

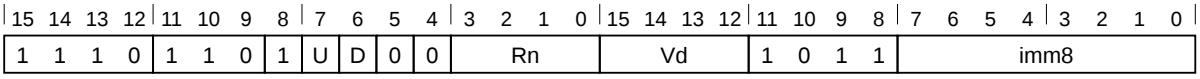
```
60                  MemA[address, 4] = cxt;
61              otherwise
62                  UNPREDICTABLE;
63
64      // If the stack pointer is being updated a fault will be raised if
65      // the limit is violated
66      if wback then
67          RSPCheck[n] = offsetAddr;
```

## C2.4.465 VSTR

Floating-point Store Register. Floating-point Store Register stores a single Floating-point Extension register to memory, using an address from a general-purpose register, with an optional offset.

### T1

*Armv8-M Floating-point Extension only or MVE*




#### T1 variant

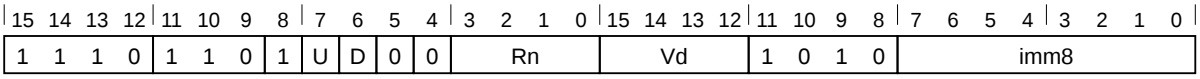
```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveOrFp);
2  if VFPSmallRegisterBank() && (D == '1') then UNDEFINED;
3  fp_size = 64;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
4  d = UInt(D:Vd);  n = UInt(Rn);
5  if n == 15 then UNPREDICTABLE;
```

### T2

*Armv8-M Floating-point Extension only or MVE*




#### T2 variant

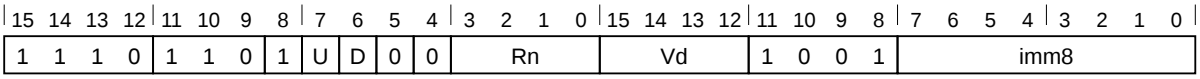
```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveOrFp);
2  fp_size = 32;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
3  d = UInt(Vd:D);  n = UInt(Rn);
4  if n == 15 then UNPREDICTABLE;
```

### T3

*Armv8.1-M Floating-point Extension only or MVE*




#### T3 variant

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveOrFp);
2  fp_size = 16;  add = (U == '1');  imm32 = ZeroExtend(imm8:'0', 32);
3  d = UInt(Vd:D);  n = UInt(Rn);
4  if n == 15 then UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| .64 | Optional data size specifiers. |
| <Dd> | The source register for a doubleword store. |
| .32 | Optional data size specifiers. |
| <Sd> | The source register for a singleword store. |
| <Rn> | Is the general-purpose base register, encoded in the "Rn" field. |
| +/- | Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:<br>- when U = 0<br>+ when U = 1 |
| <imm> | The immediate offset used for forming the address. Values are multiples of 4 in the range 0-1020. <imm> can be omitted, meaning an offset of +0. |

## Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    case fp_size of
        when 16
            MemA[address,2] = S[d]<15:0>;
        when 32
            MemA[address,4] = S[d];
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            bigEndian         = BigEndian(address, 8);
            MemA[address,4]   = if bigEndian then D[d]<63:32> else D[d]<31:0>;
            MemA[address+4,4] = if bigEndian then D[d]<31:0>  else D[d]<63:32>;
```

## C2.4.466 VSTRB, VSTRH, VSTRW

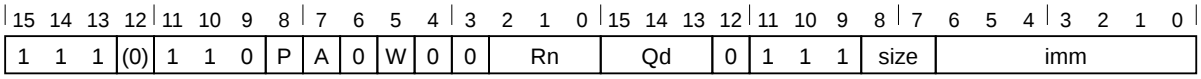
Vector Store Register. Store consecutive elements to memory from a vector register. In indexed mode, the target address is calculated from a base register offset by an immediate value. Otherwise, the base register address is used directly. The sum of the base register and the immediate value can optionally be written back to the base register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSTRB variant (Offset: P=1, W=0)

```
VSTRB<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

#### T1: VSTRB variant (Pre-indexed: P=1, W=1)

```
VSTRB<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

#### T1: VSTRB variant (Post-indexed: P=0, W=1)

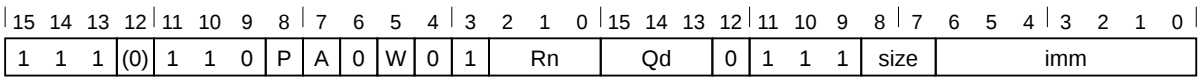
```
VSTRB<v>.<dt> Qd, [Rn], #+/-<imm>
```

### Decode for this encoding

```
if P    == '0' && W == '0' then SEE "Related encodings";
if size == '11'            then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if size == '00' then UNDEFINED;
d        = UInt(Qd);
n        = UInt(Rn);
msize    = 8;
mbytes   = msize DIV 8;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
imm32    = ZeroExtend(imm, 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = TRUE;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VSTRH variant (Offset: P=1, W=0)

```
VSTRH<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T2: VSTRH variant (Pre-indexed: P=1, W=1)

```
VSTRH<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T2: VSTRH variant (Post-indexed: P=0, W=1)

```
VSTRH<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P    == '0' && W == '0' then SEE "Related encodings";
if size == '11'            then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if size == '00' then UNDEFINED;
d        = UInt(Qd);
n        = UInt(Rn);
msize    = 16;
mbytes   = msize DIV 8;
esize    = 8 << UInt(size);
elements = 32 DIV esize;
imm32    = ZeroExtend(imm:'0', 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = TRUE;
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T5

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 0 | Rn | | | | Qd | | | | 1 | 1 | 1 | 0 | 0 | imm | | | | | | |

### T5: VSTRB variant (Offset: P=1, W=0)

```
VSTRB<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T5: VSTRB variant (Pre-indexed: P=1, W=1)

```
VSTRB<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T5: VSTRB variant (Post-indexed: P=0, W=1)

```
VSTRB<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn);
msize    = 8;
mbytes   = msize DIV 8;
esize    = msize;
elements = 32 DIV esize;
imm32    = ZeroExtend(imm, 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = TRUE;
if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
```

```
16  if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
17  if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 0 | Rn | | | | Qd | | | | 1 | 1 | 1 | 1 | 0 | 1 | imm | | | | | |

### T6: VSTRH variant (Offset: P=1, W=0)

```
VSTRH<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T6: VSTRH variant (Pre-indexed: P=1, W=1)

```
VSTRH<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T6: VSTRH variant (Post-indexed: P=0, W=1)

```
VSTRH<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
1  if P == '0' && W == '0' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' then UNDEFINED;
4  d        = UInt(D:Qd);
5  n        = UInt(Rn);
6  msize    = 16;
7  mbytes   = msize DIV 8;
8  esize    = msize;
9  elements = 32 DIV esize;
10 imm32    = ZeroExtend(imm:'0', 32);
11 index    = (P == '1');
12 add      = (A == '1');
13 wback    = (W == '1');
14 unsigned = TRUE;
15 if InITBlock()              then CONSTRAINED_UNPREDICTABLE;
16 if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
17 if Rn == '1111'             then CONSTRAINED_UNPREDICTABLE;
```

## T7

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | A | D | W | 0 | Rn | | | | Qd | | | | 1 | 1 | 1 | 1 | 1 | 0 | imm | | | | | |

### T7: VSTRW variant (Offset: P=1, W=0)

```
VSTRW<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

### T7: VSTRW variant (Pre-indexed: P=1, W=1)

```
VSTRW<v>.<dt> Qd, [Rn, #+/-<imm>]!
```

### T7: VSTRW variant (Post-indexed: P=0, W=1)

```
VSTRW<v>.<dt> Qd, [Rn], #+/-<imm>
```

## Decode for this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
CheckDecodeFaults(ExtType_Mve);
if D == '1' then UNDEFINED;
d        = UInt(D:Qd);
n        = UInt(Rn);
msize    = 32;
mbytes   = msize DIV 8;
esize    = msize;
elements = 32 DIV esize;
imm32    = ZeroExtend(imm:'00', 32);
index    = (P == '1');
add      = (A == '1');
wback    = (W == '1');
unsigned = TRUE;
if InITBlock()             then CONSTRAINED_UNPREDICTABLE;
if Rn == '1101' && W == '1' then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111'            then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
16 Encoded as size = 01
32 Encoded as size = 10

<imm> The signed immediate value that is added to base register to calculate the target address.

## Assembler symbols for T2 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
16 Encoded as size = 01
32 Encoded as size = 10

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 2.

## Assembler symbols for T5 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address.

## Assembler symbols for T6 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 2.

## Assembler symbols for T7 encodings

<dt> Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed.

<imm> The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

## Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.
<Qd> Source vector register.
<Rn> The base register for the target address.

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

offsetAddr = if add   then (R[n] + imm32) else (R[n] - imm32);
address    = if index then offsetAddr     else R[n];
address    = address + (curBeat * mbytes * elements);

for e = 0 to elements-1
    if elmtMask<e*(esize >> 3)> == '1' then
        MemA_MVE[address + (e * mbytes), mbytes] = Elem[Q[d, curBeat], e, esize]<(mbytes*8)
            -1:0>;

// The optional write back to the base register is only performed on the
// last beat of the instruction.
if wback && IsLastBeat() then
    R[n] = offsetAddr;
```

## C2.4.467 VSTRB, VSTRH, VSTRW, VSTRD (vector)

Vector Scatter Store. Store data from elements of Q[d] into a memory byte, halfword, word, or doubleword at the address contained in either:

a) A base register R[n] plus an offset contained in each element of Q[m], optionally shifted by the element size, or

b) Each element of Q[m] plus an immediate offset. The base element can optionally be written back, irrespective of predication, with that value incremented by the immediate or by the immediate scaled by the memory element size.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | 1 | D | 0 | 0 | Rn | | | | Qd | | | 0 | 1 | 1 | 1 | size | | 0 | M | 0 | Qm | | | os |

#### T1: VSTRB variant

```
VSTRB<v>.<dt> Qd, [Rn, Qm]
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' then UNDEFINED;
3  if size == '11'            then UNDEFINED;
4  d           = UInt(D:Qd);
5  m           = UInt(M:Qm);
6  n           = UInt(Rn);
7  msize       = '00';
8  esize       = 8 << UInt(size);
9  elements    = 32 DIV esize;
10 mesize      = 8;
11 add         = TRUE;
12 useReg      = TRUE;
13 scaleOffset = (os == '1');
14 wback       = FALSE;
15 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
16 if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
17 if os == '1'    then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | 1 | D | 0 | 0 | Rn | | | | Qd | | | 0 | 1 | 1 | 1 | size | | 0 | M | 1 | Qm | | | os |

#### T2: VSTRH variant

```
VSTRH<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

### Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
if size IN {'11', '00'} then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '01';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 16;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
wback       = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T3

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | 1 | D | 0 | 0 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 1 | M | 0 | Qm | | | | os |

### T3: VSTRW variant

```
VSTRW<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
if size != '10'         then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '10';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 32;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
wback       = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 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 | 1 | D | 0 | 0 | Rn | | | | Qd | | | | 0 | 1 | 1 | 1 | size | | | | 1 | M | 1 | Qm | | | | os |

### T4: VSTRD variant

```
VSTRD<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
if size != '11'         then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = UInt(Rn);
msize       = '11';
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 64;
add         = TRUE;
useReg      = TRUE;
scaleOffset = (os == '1');
wback       = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '1111' then CONSTRAINED_UNPREDICTABLE;
```

## T5

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | A | D | W | 0 | Qm | | | | (0) | Qd | | | 1 | 1 | 1 | 1 | 0 | M | imm | | | | | |

### T5: VSTRW variant (Non writeback: W=0)

```
VSTRW<v>.<dt> Qd, [Qm{, #+/-<imm>}]
```

### T5: VSTRW variant (Writeback: W=1)

```
VSTRW<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

## Decode for this encoding

```
CheckDecodeFaults(ExtType_Mve);
if D == '1' || M == '1' then UNDEFINED;
d           = UInt(D:Qd);
m           = UInt(M:Qm);
n           = integer UNKNOWN;
size        = '10';
msize       = size;
offset      = ZeroExtend(imm:Zeros(UInt(size)), 32);
esize       = 8 << UInt(size);
elements    = 32 DIV esize;
mesize      = 32;
add         = (A == '1');
useReg      = FALSE;
scaleOffset = FALSE;
wback       = (W == '1');
if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## T6

*Armv8.1-M MVE*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 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 | 1 | A | D | W | 0 | Qm | | | | (0) | Qd | | | 1 | 1 | 1 | 1 | 1 | M | imm | | | | | |

### T6: VSTRD variant (Non writeback: W=0)

```
VSTRD<v>.<dt> Qd, [Qm{, #+/-<imm>}]
```

### T6: VSTRD variant (Writeback: W=1)

```
VSTRD<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

## Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D == '1' || M == '1' then UNDEFINED;
3  d           = UInt(D:Qd);
4  m           = UInt(M:Qm);
5  n           = integer UNKNOWN;
6  size        = '11';
7  msize       = size;
8  offset      = ZeroExtend(imm:Zeros(UInt(size)), 32);
9  esize       = 8 << UInt(size);
10 elements    = 32 DIV esize;
11 mesize      = 64;
12 add         = (A == '1');
13 useReg      = FALSE;
14 scaleOffset = FALSE;
15 wback       = (W == '1');
16 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- 8 Encoded as size = 00
- 16 Encoded as size = 01
- 32 Encoded as size = 10

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

## Assembler symbols for T2 encodings

<dt> Size: indicates the size of the elements in the vector.
This parameter must be one of the following values:
- 16 Encoded as size = 01
- 32 Encoded as size = 10

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

## Assembler symbols for T3 encodings

<dt> This parameter must be the following value:
- 32 Encoded as size = 10

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

## Assembler symbols for T4 encodings

<dt> This parameter must be the following value:
- 64 Encoded as size = 11

<Qm> Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

### Assembler symbols for T5 encodings

| | |
|---|---|
| <dt> | Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed. |
| <Qm> | The base register for the target address. |
| <imm> | The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4. |

### Assembler symbols for T6 encodings

| | |
|---|---|
| <dt> | Data size. The unsigned, signed, floating and signless datatypes of the memory transfer size are allowed. |
| <Qm> | The base register for the target address. |
| <imm> | The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 8. |

### Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Qd> | Source vector register. |
| <Rn> | The base register for the target address. |
| <os> | The amount by which the vector offset is left shifted by before being added to the general-purpose base address. If the value is present it must correspond to memory transfer size (1=half word, 2=word, 3=double word).<br>This parameter must be one of the following values:<br><omitted> Encoded as os = 0<br><Offset scaled> Encoded as os = 1 |

### Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

if esize == 64 then
    // 64 bit accesses read their base address or offset from the first element
    // in each pair of 32 bit elements.
    if useReg then
        baseAddr = R[n];
        offset   = Q[m, UInt(curBeat<1>:'0')];
        if scaleOffset then
            offset = LSL(offset, UInt(msize));
    else
        baseAddr = Q[m, UInt(curBeat<1>:'0')];
    offsetAddress = if add then baseAddr + offset else baseAddr - offset;
    bigEndian     = BigEndian(offsetAddress, 8);
    address       = (if (curBeat<0> == '0') == bigEndian then offsetAddress + 4
                                                         else offsetAddress);
    if elmtMask<0> == '1' then
        MemA_MVE[address, 4] = Q[d, curBeat];
    // Address writeback is not predicated
    if wback && (curBeat<0> == '1') then
        Q[m, curBeat-1] = offsetAddress<31:0>;
else
    // 32, 16, or 8 bit accesses
    for e = 0 to (elements - 1)
        if useReg then
```

```
29              baseAddr = R[n];
30              offset   = ZeroExtend(Elem[Q[m, curBeat], e, esize], 32);
31              if scaleOffset then
32                  offset = LSL(offset, UInt(msize));
33          else
34              // 16 / 8 bit vector+immediate accesses are not supported
35              baseAddr = Q[m, curBeat];
36          address = if add then baseAddr + offset else baseAddr - offset;
37          if elmtMask<e*(esize>>3)> == '1' then
38              memValue = Elem[Q[d, curBeat], e, esize]<mesize-1:0>;
39              MemA_MVE[address, mesize DIV 8] = memValue;
40          // Address writeback is not predicated
41          if wback then
42              Elem[Q[m, curBeat], e, esize] = address<esize-1:0>;
```

## C2.4.468 VSUB (floating-point)

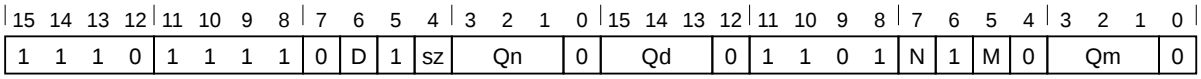
Vector Subtract. Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T1: VSUB variant

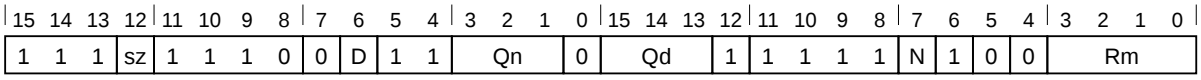
```
VSUB<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || M == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(M:Qm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = FALSE;
9  if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE with Half-precision and Single-precision Floating-point Extension*




#### T2: VSUB variant

```
VSUB<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_MveFp);
2  if D == '1' || N == '1' then UNDEFINED;
3  d          = UInt(D:Qd);
4  m          = UInt(Rm);
5  n          = UInt(N:Qn);
6  esize      = if sz == '1' then 16 else 32;
7  elements   = 32 DIV esize;
8  withScalar = TRUE;
9  if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
10 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### Assembler symbols for all encodings

<v> See C1.2.5 *Standard assembler syntax fields* on page 424.

| | |
|---|---|
| <dt> | Size: indicates the floating-point format used.<br>This parameter must be one of the following values:<br>F32 Encoded as sz = 0<br>F16 Encoded as sz = 1 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = FPSub(Elem[op1, e, esize], R[m]<esize-1:0>, FALSE);
        Elem[result, e, esize] = value;
else
    for e = 0 to elements-1
        op2   = Q[m, curBeat];
        value = FPSub(Elem[op1, e, esize], Elem[op2, e, esize], FALSE);
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.469 VSUB (vector)

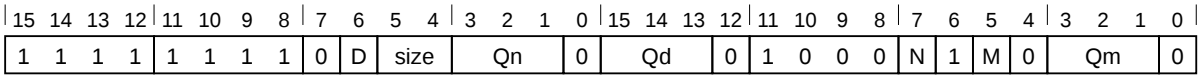
Vector Subtract. Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register. The result is then written to the destination vector register.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M MVE*




#### T1: VSUB variant

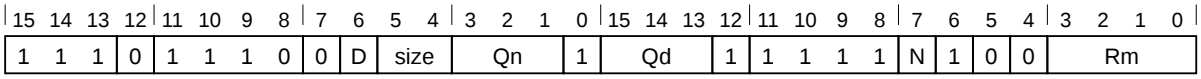
```
VSUB<v>.<dt> Qd, Qn, Qm
```

### Decode for this encoding

```
1  CheckDecodeFaults(ExtType_Mve);
2  if D    == '1' || M == '1' || N == '1' then UNDEFINED;
3  if size == '11'                        then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(M:Qm);
6  n          = UInt(N:Qn);
7  withScalar = FALSE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock() then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M MVE*




#### T2: VSUB variant

```
VSUB<v>.<dt> Qd, Qn, Rm
```

### Decode for this encoding

```
1  if size == '11' then SEE "Related encodings";
2  CheckDecodeFaults(ExtType_Mve);
3  if D == '1' || N == '1' then UNDEFINED;
4  d          = UInt(D:Qd);
5  m          = UInt(Rm);
6  n          = UInt(N:Qn);
7  withScalar = TRUE;
8  esize      = 8 << UInt(size);
9  elements   = 32 DIV esize;
10 if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
11 if Rm == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for all encodings

| | |
|---|---|
| <v> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <dt> | Size: indicates the size of the elements in the vector.<br>This parameter must be one of the following values:<br>I8 Encoded as size = 00<br>I16 Encoded as size = 01<br>I32 Encoded as size = 10 |
| <Qd> | Destination vector register. |
| <Qn> | First source vector register. |
| <Qm> | Second source vector register. |
| <Rm> | Source general-purpose register. |

## Operation for all encodings

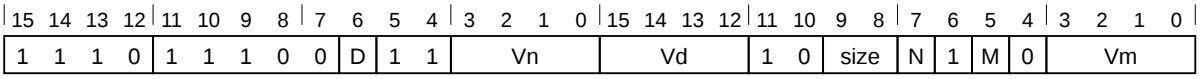
```
EncodingSpecificOperations();
ExecuteFPCheck();

(curBeat, elmtMask) = GetCurInstrBeat();

result = Zeros(32);
op1    = Q[n, curBeat];
if withScalar then
    for e = 0 to elements-1
        value = Elem[op1, e, esize] - R[m]<esize-1:0>;
        Elem[result, e, esize] = value;
else
    op2 = Q[m, curBeat];
    for e = 0 to elements-1
        value = Elem[op1, e, esize] - Elem[op2, e, esize];
        Elem[result, e, esize] = value;

for e = 0 to 3
    if elmtMask<e> == '1' then
        Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

## C2.4.470 VSUB

Floating-point Subtract. Floating-point Subtract subtracts one floating-point register value from another floating-point register value, and places the results in the destination floating-point register.

### T2

*Armv8-M Floating-point Extension only*, size == 11 UNDEFINED in single-precision only implementations.




**Half-precision scalar variant**

***Armv8.1-M Floating-point Extension only.***

**Applies when `size == 01`.**

```
VSUB{<c>}{<q>}.F16 {<Sd>,} <Sn>, <Sm>
```

**Single-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 10`.**

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

**Double-precision scalar variant**

***Armv8-M Floating-point Extension only.***

**Applies when `size == 11`.**

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

### Decode for this encoding

```
1  dp_operation = (size == '11');
2  CheckFPDecodeFaults(size);
3  if VFPSmallRegisterBank() && dp_operation && (M == '1' || N == '1' || D == '1') then
       UNDEFINED;
4  if size == '01' && InITBlock() then UNPREDICTABLE;
5  d = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
6  n = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
7  m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <Sd> | Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field. |
| <Sn> | Is the 32-bit name of the first floating-point source register, encoded in the "Vn:N" field. |
| <Sm> | Is the 32-bit name of the second floating-point source register, encoded in the "Vm:M" field. |
| <Dd> | Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field. |
| <Dn> | Is the 64-bit name of the first floating-point source register, encoded in the "N:Vn" field. |
| <Dm> | Is the 64-bit name of the second floating-point source register, encoded in the "M:Vm" field. |

### Operation for all encodings

```
1  if ConditionPassed() then
```

```
    EncodingSpecificOperations();
    ExecuteFPCheck();
    case size of
        when '01'
            S[d] = Zeros(16) : FPSub(S[n]<15:0>, S[m]<15:0>, TRUE);
        when '10'
            S[d] = FPSub(S[n], S[m], TRUE);
        when '11'
            D[d] = FPSub(D[n], D[m], TRUE);
```

## C2.4.471 WFE

Wait For Event. Wait For Event is a hint instruction. If the Event Register is clear, it suspends execution in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, exception or other event occurs.

This is a NOP-compatible hint.

### T1

*Armv8-M*

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

#### T1 variant

```
WFE{<c>}{<q>}
```

### Decode for this encoding

```
// No additional decoding required
```

### T2

*Armv8-M Main Extension only*

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

#### T2 variant

```
WFE{<c>}.W
```

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

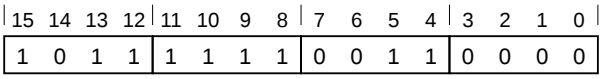
## C2.4.472 WFI

Wait For Interrupt. Wait For Interrupt is a hint instruction. It suspends execution, in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, asynchronous exception or other event occurs.

This is a NOP-compatible hint.

### T1

*Armv8-M*




#### T1 variant

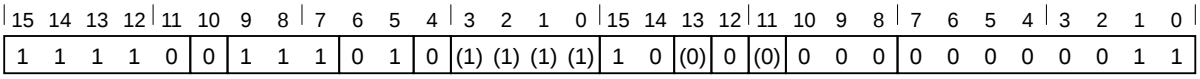
`WFI{<c>}{<q>}`

### Decode for this encoding

```
// No additional decoding required
```

### T2

*Armv8-M Main Extension only*




#### T2 variant

`WFI{<c>}.W`

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

### Assembler symbols for all encodings

| | |
|---|---|
| <c> | See C1.2.5 *Standard assembler syntax fields* on page 424. |
| <q> | See C1.2.5 *Standard assembler syntax fields* on page 424. |

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

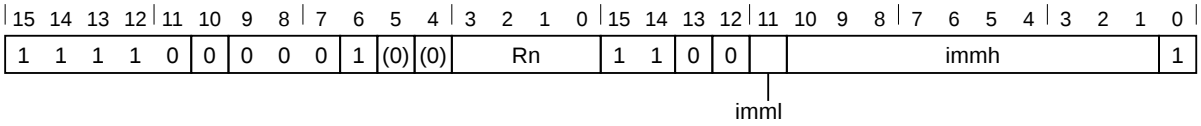
## C2.4.473 WLS, DLS, WLSTP, DLSTP

While Loop Start, Do Loop Start, While Loop Start with Tail Predication, Do Loop Start with Tail Predication. This instruction partially sets up a loop. A LE or LETP (Loop End) instruction completes the setup. The base variants of this instruction (WLS and DLS) set LR to the number of loop iterations to be performed, whereas the TP variants of this instruction set LR to the number of vector-elements that must be processed. For the TP variants, if the number of elements required is not a multiple of the vector length then the appropriate number of vector elements will be predicated on the last iteration of the loop. When using WLS or WLSTP, if the number of iterations required is zero, then these instructions branch to the label specified. Each loop start instruction is normally used with a matching LE or LETP instruction.

This instruction is not permitted in an IT block.

### T1

*Armv8.1-M Low Overhead Branch Extension*




#### T1: WLS variant

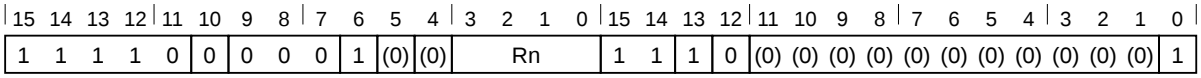
```
WLS LR, Rn, <label>
```

### Decode for this encoding

```
if !HaveLOBExt() then UNDEFINED;
n           = UInt(Rn);
tSize       = 4<2:0>; // No truncation. Set size to full vector length
imm32       = ZeroExtend(immh:imml:'0', 32);
isWhileLoop = TRUE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T2

*Armv8.1-M Low Overhead Branch Extension*




#### T2: DLS variant

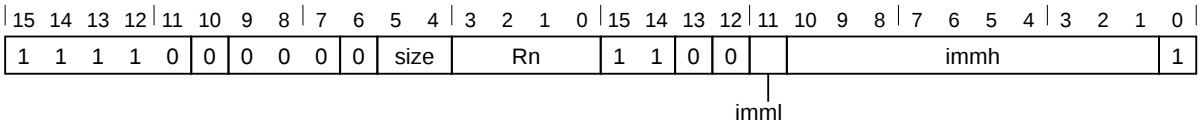
```
DLS LR, Rn
```

### Decode for this encoding

```
if !HaveLOBExt() then UNDEFINED;
n           = UInt(Rn);
tSize       = 4<2:0>; // No truncation. Set size to full vector length
imm32       = Zeros(32);
isWhileLoop = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '11x1' then CONSTRAINED_UNPREDICTABLE;
```

### T3

*Armv8.1-M Low Overhead Branch Extension and MVE*




### T3: WLSTP variant

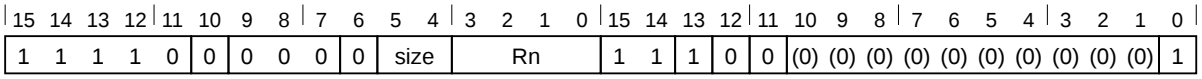
`WLSTP.<size> LR, Rn, <label>`

## Decode for this encoding

```
if Rn == '1111'  then SEE "Related encodings";
if !HaveLOBExt() then UNDEFINED;
if !HaveMve()    then UNDEFINED;
HandleException(CheckCPEnabled(10));
n           = UInt(Rn);
tSize       = '0':size;
imm32       = ZeroExtend(immh:imml:'0', 32);
isWhileLoop = TRUE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## T4

*Armv8.1-M Low Overhead Branch Extension and MVE*




### T4: DLSTP variant

`DLSTP.<size> LR, Rn`

## Decode for this encoding

```
if Rn == '1111'  then SEE "Related encodings";
if !HaveLOBExt() then UNDEFINED;
if !HaveMve()    then UNDEFINED;
HandleException(CheckCPEnabled(10));
n           = UInt(Rn);
tSize       = '0':size;
imm32       = Zeros(32);
isWhileLoop = FALSE;
if InITBlock()  then CONSTRAINED_UNPREDICTABLE;
if Rn == '1101' then CONSTRAINED_UNPREDICTABLE;
```

## Assembler symbols for T1 encodings

| | |
|---|---|
| <LR> | LR is used to hold the iteration counter of the loop, this instruction must always use this register. |
| <Rn> | The register holding the number of loop iterations to perform. |
| <label> | Specifies the label of the instruction to branch to if no loop iterations are required. |

## Assembler symbols for T2 encodings

| | |
|---|---|
| <LR> | LR is used to hold the iteration counter of the loop, this instruction must always use this register. |

<Rn> The register holding the number of loop iterations to perform.

### Assembler symbols for T3 encodings

<LR> LR is used to hold the number of elements to process, this instruction must always use this register.
<Rn> The register holding the number of elements to process.
<label> Specifies the label of the instruction after the loop (the first instruction after the LE).

### Assembler symbols for T4 encodings

<LR> LR is used to hold the number of elements to process, this instruction must always use this register.
<Rn> The register holding the number of elements to process.

### Assembler symbols for all encodings

<size> The size of the elements in the vector to process. This value is stored in the FPSCR.LTPSIZE field, and causes tail predication to be applied on the last iteration of the loop.
This parameter must be one of the following values:

| | | |
|---|---|---|
| 8 | Encoded as | size = 00 |
| 16 | Encoded as | size = 01 |
| 32 | Encoded as | size = 10 |
| 64 | Encoded as | size = 11 |

### Operation for all encodings

```
EncodingSpecificOperations();

count = R[n];
if isWhileLoop && count == Zeros(32) then
    BranchTo(PC + imm32);
else
    // To avoid creating unnecessary FP context, the LTPSIZE is only set if
    // tail predication is being used.
    if tSize != 4<2:0> then
        ExecuteFPCheck();
        FPSCR.LTPSIZE = tSize;
    // Set up the new iteration count
    LR = count;
```

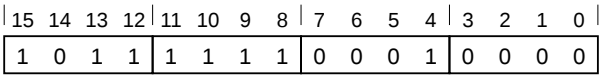
## C2.4.474 YIELD

Yield hint. Yield is a hint instruction. It enables software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spinlock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

This is a NOP-compatible hint.

### T1

*Armv8-M*



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

#### T1 variant

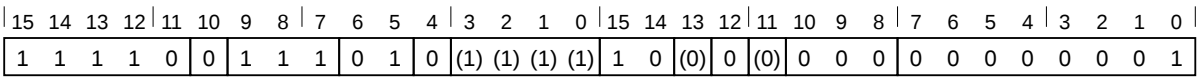
`YIELD{<c>}{<q>}`

### Decode for this encoding

```
// No additional decoding required
```

### T2

*Armv8-M Main Extension only*



| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | (1) | (1) | (1) | (1) | 1 | 0 | (0) | 0 | (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

#### T2 variant

`YIELD{<c>}.W`

### Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

### Assembler symbols for all encodings

<c> See C1.2.5 *Standard assembler syntax fields* on page 424.
<q> See C1.2.5 *Standard assembler syntax fields* on page 424.

### Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

# Part D
## Armv8-M Registers

# Chapter D1
# Register Specification

This chapter specifies the Armv8-M registers. It contains the following sections:

# D1.1 Register index

| Address | Component |
|---|---|
| – | Special and general-purpose registers |
| – | Payloads |
| 0xE0000000 | Instrumentation Macrocell |
| 0xE0001000 | Data Watchpoint and Trace |
| 0xE0002000 | Flash Patch and Breakpoint |
| 0xE0003000 | Performance Monitoring Unit |
| 0xE0005000 | Reliability, Availability and Serviceability Extension Fault Status Register |
| 0xE000E004 | Implementation Control Block |
| 0xE000E010 | SysTick Timer |
| 0xE000E100 | Nested Vectored Interrupt Controller |
| 0xE000ECFC | System Control Block |
| 0xE000ED90 | Memory Protection Unit |
| 0xE000EDD0 | Security Attribution Unit |
| 0xE000EDF0 | Debug Control Block |
| 0xE000EF00 | Software Interrupt Generation |
| 0xE000EF04 | Reliability, Availability and Serviceability Extension Fault Status Register |
| 0xE000EF34 | Floating-Point Extension |
| 0xE000EF50 | Cache Maintenance Operations |
| 0xE000EFB0 | Debug Identification Block |
| 0xE002E004 | Implementation Control Block (NS alias) |
| 0xE002E010 | SysTick Timer (NS alias) |
| 0xE002E100 | Nested Vectored Interrupt Controller (NS alias) |
| 0xE002ECFC | System Control Block (NS alias) |
| 0xE002ED90 | Memory Protection Unit (NS alias) |
| 0xE002EDF0 | Debug Control Block (NS alias) |
| 0xE002EF00 | Software Interrupt Generation (NS alias) |
| 0xE002EF04 | Reliability, Availability and Serviceability Extension Fault Status Register (NS Alias) |
| 0xE002EF34 | Floating-Point Extension (NS alias) |
| 0xE002EF50 | Cache Maintenance Operations (NS alias) |
| 0xE002EFB0 | Debug Identification Block (NS alias) |
| 0xE0040000 | Trace Port Interface Unit |

## D1.1.1 Special and general-purpose registers

| Name | Description |
|---|---|
| APSR | Application Program Status Register |
| BASEPRI | Base Priority Mask Register |
| CONTROL | Control Register |
| EPSR | Execution Program Status Register |
| FAULTMASK | Fault Mask Register |
| FPSCR | Floating-point Status and Control Register |
| IPSR | Interrupt Program Status Register |
| LO_BRANCH_INFO | Loop and branch tracking information |
| LR | Link Register |
| MSPLIM | Main Stack Pointer Limit Register |
| PC | Program Counter |
| PRIMASK | Exception Mask Register |
| PSPLIM | Process Stack Pointer Limit Register |

| Name | Description |
|---|---|
| Rn | General-Purpose Register *n* |
| SP | Current Stack Pointer Register |
| SP | Stack Pointer (Non-secure) |
| VPR | Vector Predication Status and Control Register |
| XPSR | Combined Program Status Registers |

### D1.1.2 Payloads

| Name | Description |
|---|---|
| EXC_RETURN | Exception Return Payload |
| FNC_RETURN | Function Return Payload |
| FPCXT | Floating-point context payload |
| MAIR_ATTR | Memory Attribute Indirection Register Attributes |
| RETPSR | Combined Exception Return Program Status Registers |
| TT_RESP | Test Target Response Payload |

### D1.1.3 Instrumentation Macrocell

| Address | Register | Description |
|---|---|---|
| 0xE0000000 | ITM_STIMn | ITM Stimulus Port Register *n* |
| 0xE0000E00 | ITM_TERn | ITM Trace Enable Register *n* |
| 0xE0000E40 | ITM_TPR | ITM Trace Privilege Register |
| 0xE0000E80 | ITM_TCR | ITM Trace Control Register |
| 0xE0000FB0 | ITM_LAR | ITM Software Lock Access Register |
| 0xE0000FB4 | ITM_LSR | ITM Software Lock Status Register |
| 0xE0000FBC | ITM_DEVARCH | ITM Device Architecture Register |
| 0xE0000FCC | ITM_DEVTYPE | ITM Device Type Register |
| 0xE0000FD0 | ITM_PIDR4 | ITM Peripheral Identification Register 4 |
| 0xE0000FD4 | ITM_PIDR5 | ITM Peripheral Identification Register 5 |
| 0xE0000FD8 | ITM_PIDR6 | ITM Peripheral Identification Register 6 |
| 0xE0000FDC | ITM_PIDR7 | ITM Peripheral Identification Register 7 |
| 0xE0000FE0 | ITM_PIDR0 | ITM Peripheral Identification Register 0 |
| 0xE0000FE4 | ITM_PIDR1 | ITM Peripheral Identification Register 1 |
| 0xE0000FE8 | ITM_PIDR2 | ITM Peripheral Identification Register 2 |
| 0xE0000FEC | ITM_PIDR3 | ITM Peripheral Identification Register 3 |
| 0xE0000FF0 | ITM_CIDR0 | ITM Component Identification Register 0 |
| 0xE0000FF4 | ITM_CIDR1 | ITM Component Identification Register 1 |
| 0xE0000FF8 | ITM_CIDR2 | ITM Component Identification Register 2 |
| 0xE0000FFC | ITM_CIDR3 | ITM Component Identification Register 3 |

### D1.1.4 Data Watchpoint and Trace

| Address | Register | Description |
|---|---|---|
| 0xE0001000 | DWT_CTRL | DWT Control Register |
| 0xE0001004 | DWT_CYCCNT | DWT Cycle Count Register |
| 0xE0001008 | DWT_CPICNT | DWT CPI Count Register |

| Address | Register | Description |
|---|---|---|
| 0xE000100C | DWT_EXCCNT | DWT Exception Overhead Count Register |
| 0xE0001010 | DWT_SLEEPCNT | DWT Sleep Count Register |
| 0xE0001014 | DWT_LSUCNT | DWT LSU Count Register |
| 0xE0001018 | DWT_FOLDCNT | DWT Folded Instruction Count Register |
| 0xE000101C | DWT_PCSR | DWT Program Counter Sample Register |
| 0xE0001020 | DWT_COMPn | DWT Comparator Register *n* |
| 0xE0001028 | DWT_FUNCTIONn | DWT Comparator Function Register *n* |
| 0xE000102C | DWT_VMASKn | DWT Comparator Value Mask Register *n* |
| 0xE0001FB0 | DWT_LAR | DWT Software Lock Access Register |
| 0xE0001FB4 | DWT_LSR | DWT Software Lock Status Register |
| 0xE0001FBC | DWT_DEVARCH | DWT Device Architecture Register |
| 0xE0001FCC | DWT_DEVTYPE | DWT Device Type Register |
| 0xE0001FD0 | DWT_PIDR4 | DWT Peripheral Identification Register 4 |
| 0xE0001FD4 | DWT_PIDR5 | DWT Peripheral Identification Register 5 |
| 0xE0001FD8 | DWT_PIDR6 | DWT Peripheral Identification Register 6 |
| 0xE0001FDC | DWT_PIDR7 | DWT Peripheral Identification Register 7 |
| 0xE0001FE0 | DWT_PIDR0 | DWT Peripheral Identification Register 0 |
| 0xE0001FE4 | DWT_PIDR1 | DWT Peripheral Identification Register 1 |
| 0xE0001FE8 | DWT_PIDR2 | DWT Peripheral Identification Register 2 |
| 0xE0001FEC | DWT_PIDR3 | DWT Peripheral Identification Register 3 |
| 0xE0001FF0 | DWT_CIDR0 | DWT Component Identification Register 0 |
| 0xE0001FF4 | DWT_CIDR1 | DWT Component Identification Register 1 |
| 0xE0001FF8 | DWT_CIDR2 | DWT Component Identification Register 2 |
| 0xE0001FFC | DWT_CIDR3 | DWT Component Identification Register 3 |

### D1.1.5 Flash Patch and Breakpoint

| Address | Register | Description |
|---|---|---|
| 0xE0002000 | FP_CTRL | Flash Patch Control Register |
| 0xE0002004 | FP_REMAP | Flash Patch Remap Register |
| 0xE0002008 | FP_COMPn | Flash Patch Comparator Register *n* |
| 0xE0002FB0 | FP_LAR | FPB Software Lock Access Register |
| 0xE0002FB4 | FP_LSR | FPB Software Lock Status Register |
| 0xE0002FBC | FP_DEVARCH | FPB Device Architecture Register |
| 0xE0002FCC | FP_DEVTYPE | FPB Device Type Register |
| 0xE0002FD0 | FP_PIDR4 | FP Peripheral Identification Register 4 |
| 0xE0002FD4 | FP_PIDR5 | FP Peripheral Identification Register 5 |
| 0xE0002FD8 | FP_PIDR6 | FP Peripheral Identification Register 6 |
| 0xE0002FDC | FP_PIDR7 | FP Peripheral Identification Register 7 |
| 0xE0002FE0 | FP_PIDR0 | FP Peripheral Identification Register 0 |
| 0xE0002FE4 | FP_PIDR1 | FP Peripheral Identification Register 1 |
| 0xE0002FE8 | FP_PIDR2 | FP Peripheral Identification Register 2 |
| 0xE0002FEC | FP_PIDR3 | FP Peripheral Identification Register 3 |
| 0xE0002FF0 | FP_CIDR0 | FP Component Identification Register 0 |
| 0xE0002FF4 | FP_CIDR1 | FP Component Identification Register 1 |
| 0xE0002FF8 | FP_CIDR2 | FP Component Identification Register 2 |
| 0xE0002FFC | FP_CIDR3 | FP Component Identification Register 3 |

### D1.1.6 Performance Monitoring Unit

| Address | Register | Description |
|---|---|---|
| 0xE0003000 | PMU_EVCNTRn | Performance Monitoring Unit Event Counter Register |
| 0xE000307C | PMU_CCNTR | Performance Monitoring Unit Cycle Counter Register |
| 0xE0003400 | PMU_EVTYPERn | Performance Monitoring Unit Event Type and Filter Register |
| 0xE000347C | PMU_CCFILTR | Performance Monitoring Unit Cycle Counter Filter Register |
| 0xE0003C00 | PMU_CNTENSET | Performance Monitoring Unit Count Enable Set Register |
| 0xE0003C20 | PMU_CNTENCLR | Performance Monitoring Unit Count Enable Clear Register |
| 0xE0003C40 | PMU_INTENSET | Performance Monitoring Unit Interrupt Enable Set Register |
| 0xE0003C60 | PMU_INTENCLR | Performance Monitoring Unit Interrupt Enable Clear Register |
| 0xE0003C80 | PMU_OVSCLR | Performance Monitoring Unit Overflow Flag Status Clear Register |
| 0xE0003CA0 | PMU_SWINC | Performance Monitoring Unit Software Increment Register |
| 0xE0003CC0 | PMU_OVSSET | Performance Monitoring Unit Overflow Flag Status Set Register |
| 0xE0003E00 | PMU_TYPE | Performance Monitoring Unit Type Register |
| 0xE0003E04 | PMU_CTRL | Performance Monitoring Unit Control Register |
| 0xE0003FB8 | PMU_AUTHSTATUS | Performance Monitoring Unit Authentication Status Register |
| 0xE0003FBC | PMU_DEVARCH | Performance Monitoring Unit Device Architecture Register |
| 0xE0003FCC | PMU_DEVTYPE | Performance Monitoring Unit Device Type Register |
| 0xE0003FD0 | PMU_PIDR4 | Performance Monitoring Unit Peripheral Identification Register 4 |
| 0xE0003FE0 | PMU_PIDR0 | Performance Monitoring Unit Peripheral Identification Register 0 |
| 0xE0003FE4 | PMU_PIDR1 | Performance Monitoring Unit Peripheral Identification Register 1 |
| 0xE0003FE8 | PMU_PIDR2 | Performance Monitoring Unit Peripheral Identification Register 2 |
| 0xE0003FEC | PMU_PIDR3 | Performance Monitoring Unit Peripheral Identification Register 3 |
| 0xE0003FF0 | PMU_CIDR0 | Performance Monitoring Unit Component Identification Register 0 |
| 0xE0003FF4 | PMU_CIDR1 | Performance Monitoring Unit Component Identification Register 1 |
| 0xE0003FF8 | PMU_CIDR2 | Performance Monitoring Unit Component Identification Register 2 |
| 0xE0003FFC | PMU_CIDR3 | Performance Monitoring Unit Component Identification Register 3 |

### D1.1.7 Reliability, Availability and Serviceability Extension Fault Status Register

| Address | Register | Description |
|---|---|---|
| 0xE0005000 | ERRFRn | Error Record Feature Register *n* |
| 0xE0005008 | ERRCTRLn | Error Record Control Register *n* |
| 0xE0005010 | ERRSTATUSn | Error Record Primary Status Register *n* |
| 0xE0005018 | ERRADDRn | Error Record Address Register *n* |
| 0xE000501C | ERRADDR2n | Error Record Address 2 Register *n* |
| 0xE0005020 | ERRMISC0n | Error Record Miscellaneous 0 Register *n* |
| 0xE0005024 | ERRMISC1n | Error Record Miscellaneous 1 Register *n* |
| 0xE0005028 | ERRMISC2n | Error Record Miscellaneous 2 Register *n* |
| 0xE000502C | ERRMISC3n | Error Record Miscellaneous 3 Register *n* |
| 0xE0005030 | ERRMISC4n | Error Record Miscellaneous 4 Register *n* |
| 0xE0005034 | ERRMISC5n | Error Record Miscellaneous 5 Register *n* |
| 0xE0005038 | ERRMISC6n | Error Record Miscellaneous 6 Register *n* |
| 0xE000503C | ERRMISC7n | Error Record Miscellaneous 7 Register *n* |
| 0xE0005E00 | ERRGSRn | RAS Fault Group Status Register |
| 0xE0005E10 | ERRIIDR | Error Implementer ID Register |
| 0xE0005FC8 | ERRDEVID | Error Record Device ID Register |

### D1.1.8 Implementation Control Block

| Address | Register | Description |
|---|---|---|
| 0xE000E004 | ICTR | Interrupt Controller Type Register |
| 0xE000E008 | ACTLR | Auxiliary Control Register |
| 0xE000E00C | CPPWR | Coprocessor Power Control Register |

### D1.1.9 SysTick Timer

| Address | Register | Description |
|---|---|---|
| 0xE000E010 | SYST_CSR | SysTick Control and Status Register |
| 0xE000E014 | SYST_RVR | SysTick Reload Value Register |
| 0xE000E018 | SYST_CVR | SysTick Current Value Register |
| 0xE000E01C | SYST_CALIB | SysTick Calibration Value Register |

### D1.1.10 Nested Vectored Interrupt Controller

| Address | Register | Description |
|---|---|---|
| 0xE000E100 | NVIC_ISERn | Interrupt Set Enable Register *n* |
| 0xE000E180 | NVIC_ICERn | Interrupt Clear Enable Register *n* |
| 0xE000E200 | NVIC_ISPRn | Interrupt Set Pending Register *n* |
| 0xE000E280 | NVIC_ICPRn | Interrupt Clear Pending Register *n* |
| 0xE000E300 | NVIC_IABRn | Interrupt Active Bit Register *n* |
| 0xE000E380 | NVIC_ITNSn | Interrupt Target Non-secure Register *n* |
| 0xE000E400 | NVIC_IPRn | Interrupt Priority Register *n* |

### D1.1.11 System Control Block

| Address | Register | Description |
|---|---|---|
| 0xE000ECFC | REVIDR | Revision ID Register |
| 0xE000ED00 | CPUID | CPUID Base Register |
| 0xE000ED04 | ICSR | Interrupt Control and State Register |
| 0xE000ED08 | VTOR | Vector Table Offset Register |
| 0xE000ED0C | AIRCR | Application Interrupt and Reset Control Register |
| 0xE000ED10 | SCR | System Control Register |
| 0xE000ED14 | CCR | Configuration and Control Register |
| 0xE000ED18 | SHPR1 | System Handler Priority Register 1 |
| 0xE000ED1C | SHPR2 | System Handler Priority Register 2 |
| 0xE000ED20 | SHPR3 | System Handler Priority Register 3 |
| 0xE000ED24 | SHCSR | System Handler Control and State Register |
| 0xE000ED28 | MMFSR | MemManage Fault Status Register |
| 0xE000ED28 | CFSR | Configurable Fault Status Register |
| 0xE000ED29 | BFSR | BusFault Status Register |
| 0xE000ED2A | UFSR | UsageFault Status Register |
| 0xE000ED2C | HFSR | HardFault Status Register |
| 0xE000ED30 | DFSR | Debug Fault Status Register |
| 0xE000ED34 | MMFAR | MemManage Fault Address Register |
| 0xE000ED38 | BFAR | BusFault Address Register |
| 0xE000ED3C | AFSR | Auxiliary Fault Status Register |

| Address | Register | Description |
|---|---|---|
| 0xE000ED40 | ID_PFR0 | Processor Feature Register 0 |
| 0xE000ED44 | ID_PFR1 | Processor Feature Register 1 |
| 0xE000ED48 | ID_DFR0 | Debug Feature Register 0 |
| 0xE000ED4C | ID_AFR0 | Auxiliary Feature Register 0 |
| 0xE000ED50 | ID_MMFR0 | Memory Model Feature Register 0 |
| 0xE000ED54 | ID_MMFR1 | Memory Model Feature Register 1 |
| 0xE000ED58 | ID_MMFR2 | Memory Model Feature Register 2 |
| 0xE000ED5C | ID_MMFR3 | Memory Model Feature Register 3 |
| 0xE000ED60 | ID_ISAR0 | Instruction Set Attribute Register 0 |
| 0xE000ED64 | ID_ISAR1 | Instruction Set Attribute Register 1 |
| 0xE000ED68 | ID_ISAR2 | Instruction Set Attribute Register 2 |
| 0xE000ED6C | ID_ISAR3 | Instruction Set Attribute Register 3 |
| 0xE000ED70 | ID_ISAR4 | Instruction Set Attribute Register 4 |
| 0xE000ED74 | ID_ISAR5 | Instruction Set Attribute Register 5 |
| 0xE000ED78 | CLIDR | Cache Level ID Register |
| 0xE000ED7C | CTR | Cache Type Register |
| 0xE000ED80 | CCSIDR | Current Cache Size ID register |
| 0xE000ED84 | CSSELR | Cache Size Selection Register |
| 0xE000ED88 | CPACR | Coprocessor Access Control Register |
| 0xE000ED8C | NSACR | Non-secure Access Control Register |

### D1.1.12 Memory Protection Unit

| Address | Register | Description |
|---|---|---|
| 0xE000ED90 | MPU_TYPE | MPU Type Register |
| 0xE000ED94 | MPU_CTRL | MPU Control Register |
| 0xE000ED98 | MPU_RNR | MPU Region Number Register |
| 0xE000ED9C | MPU_RBAR | MPU Region Base Address Register |
| 0xE000EDA0 | MPU_RLAR | MPU Region Limit Address Register |
| 0xE000EDA4 | MPU_RBAR_An | MPU Region Base Address Register Alias *n* |
| 0xE000EDA8 | MPU_RLAR_An | MPU Region Limit Address Register Alias *n* |
| 0xE000EDC0 | MPU_MAIR0 | MPU Memory Attribute Indirection Register 0 |
| 0xE000EDC4 | MPU_MAIR1 | MPU Memory Attribute Indirection Register 1 |

### D1.1.13 Security Attribution Unit

| Address | Register | Description |
|---|---|---|
| 0xE000EDD0 | SAU_CTRL | SAU Control Register |
| 0xE000EDD4 | SAU_TYPE | SAU Type Register |
| 0xE000EDD8 | SAU_RNR | SAU Region Number Register |
| 0xE000EDDC | SAU_RBAR | SAU Region Base Address Register |
| 0xE000EDE0 | SAU_RLAR | SAU Region Limit Address Register |
| 0xE000EDE4 | SFSR | Secure Fault Status Register |
| 0xE000EDE8 | SFAR | Secure Fault Address Register |

### D1.1.14 Debug Control Block

| Address | Register | Description |
|---|---|---|
| 0xE000EDF0 | DHCSR | Debug Halting Control and Status Register |
| 0xE000EDF4 | DCRSR | Debug Core Register Select Register |
| 0xE000EDF8 | DCRDR | Debug Core Register Data Register |
| 0xE000EDFC | DEMCR | Debug Exception and Monitor Control Register |
| 0xE000EE00 | DSCEMCR | Debug Set Clear Exception and Monitor Control Register |
| 0xE000EE04 | DAUTHCTRL | Debug Authentication Control Register |
| 0xE000EE08 | DSCSR | Debug Security Control and Status Register |

### D1.1.15 Software Interrupt Generation

| Address | Register | Description |
|---|---|---|
| 0xE000EF00 | STIR | Software Triggered Interrupt Register |

### D1.1.16 Reliability, Availability and Serviceability Extension Fault Status Register

| Address | Register | Description |
|---|---|---|
| 0xE000EF04 | RFSR | RAS Fault Status Register |

### D1.1.17 Floating-Point Extension

| Address | Register | Description |
|---|---|---|
| 0xE000EF34 | FPCCR | Floating-Point Context Control Register |
| 0xE000EF38 | FPCAR | Floating-Point Context Address Register |
| 0xE000EF3C | FPDSCR | Floating-Point Default Status Control Register |
| 0xE000EF40 | MVFR0 | Media and VFP Feature Register 0 |
| 0xE000EF44 | MVFR1 | Media and VFP Feature Register 1 |
| 0xE000EF48 | MVFR2 | Media and VFP Feature Register 2 |

### D1.1.18 Cache Maintenance Operations

| Address | Register | Description |
|---|---|---|
| 0xE000EF50 | ICIALLU | Instruction Cache Invalidate All to PoU |
| 0xE000EF58 | ICIMVAU | Instruction Cache line Invalidate by Address to PoU |
| 0xE000EF5C | DCIMVAC | Data Cache line Invalidate by Address to PoC |
| 0xE000EF60 | DCISW | Data Cache line Invalidate by Set/Way |
| 0xE000EF64 | DCCMVAU | Data Cache line Clean by address to PoU |
| 0xE000EF68 | DCCMVAC | Data Cache line Clean by Address to PoC |
| 0xE000EF6C | DCCSW | Data Cache Clean line by Set/Way |
| 0xE000EF70 | DCCIMVAC | Data Cache line Clean and Invalidate by Address to PoC |
| 0xE000EF74 | DCCISW | Data Cache line Clean and Invalidate by Set/Way |
| 0xE000EF78 | BPIALL | Branch Predictor Invalidate All |

### D1.1.19 Debug Identification Block

| Address | Register | Description |
|---|---|---|
| 0xE000EFB0 | DLAR | SCS Software Lock Access Register |
| 0xE000EFB4 | DLSR | SCS Software Lock Status Register |
| 0xE000EFB8 | DAUTHSTATUS | Debug Authentication Status Register |
| 0xE000EFBC | DDEVARCH | SCS Device Architecture Register |
| 0xE000EFCC | DDEVTYPE | SCS Device Type Register |
| 0xE000EFD0 | DPIDR4 | SCS Peripheral Identification Register 4 |
| 0xE000EFD4 | DPIDR5 | SCS Peripheral Identification Register 5 |
| 0xE000EFD8 | DPIDR6 | SCS Peripheral Identification Register 6 |
| 0xE000EFDC | DPIDR7 | SCS Peripheral Identification Register 7 |
| 0xE000EFE0 | DPIDR0 | SCS Peripheral Identification Register 0 |
| 0xE000EFE4 | DPIDR1 | SCS Peripheral Identification Register 1 |
| 0xE000EFE8 | DPIDR2 | SCS Peripheral Identification Register 2 |
| 0xE000EFEC | DPIDR3 | SCS Peripheral Identification Register 3 |
| 0xE000EFF0 | DCIDR0 | SCS Component Identification Register 0 |
| 0xE000EFF4 | DCIDR1 | SCS Component Identification Register 1 |
| 0xE000EFF8 | DCIDR2 | SCS Component Identification Register 2 |
| 0xE000EFFC | DCIDR3 | SCS Component Identification Register 3 |

### D1.1.20 Implementation Control Block (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002E004 | ICTR | Interrupt Controller Type Register (NS) |
| 0xE002E008 | ACTLR | Auxiliary Control Register (NS) |
| 0xE002E00C | CPPWR | Coprocessor Power Control Register (NS) |

### D1.1.21 SysTick Timer (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002E010 | SYST_CSR | SysTick Control and Status Register (NS) |
| 0xE002E014 | SYST_RVR | SysTick Reload Value Register (NS) |
| 0xE002E018 | SYST_CVR | SysTick Current Value Register (NS) |
| 0xE002E01C | SYST_CALIB | SysTick Calibration Value Register (NS) |

### D1.1.22 Nested Vectored Interrupt Controller (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002E100 | NVIC_ISERn | Interrupt Set Enable Register *n* (NS) |
| 0xE002E180 | NVIC_ICERn | Interrupt Clear Enable Register *n* (NS) |
| 0xE002E200 | NVIC_ISPRn | Interrupt Set Pending Register *n* (NS) |
| 0xE002E280 | NVIC_ICPRn | Interrupt Clear Pending Register *n* (NS) |
| 0xE002E300 | NVIC_IABRn | Interrupt Active Bit Register *n* (NS) |
| 0xE002E400 | NVIC_IPRn | Interrupt Priority Register *n* (NS) |

### D1.1.23 System Control Block (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002ECFC | REVIDR | Revision ID Register (NS) |
| 0xE002ED00 | CPUID | CPUID Base Register (NS) |
| 0xE002ED04 | ICSR | Interrupt Control and State Register (NS) |
| 0xE002ED08 | VTOR | Vector Table Offset Register (NS) |
| 0xE002ED0C | AIRCR | Application Interrupt and Reset Control Register (NS) |
| 0xE002ED10 | SCR | System Control Register (NS) |
| 0xE002ED14 | CCR | Configuration and Control Register (NS) |
| 0xE002ED18 | SHPR1 | System Handler Priority Register 1 (NS) |
| 0xE002ED1C | SHPR2 | System Handler Priority Register 2 (NS) |
| 0xE002ED20 | SHPR3 | System Handler Priority Register 3 (NS) |
| 0xE002ED24 | SHCSR | System Handler Control and State Register (NS) |
| 0xE002ED28 | CFSR | Configurable Fault Status Register (NS) |
| 0xE002ED28 | MMFSR | MemManage Fault Status Register (NS) |
| 0xE002ED29 | BFSR | BusFault Status Register (NS) |
| 0xE002ED2A | UFSR | UsageFault Status Register (NS) |
| 0xE002ED2C | HFSR | HardFault Status Register (NS) |
| 0xE002ED30 | DFSR | Debug Fault Status Register (NS) |
| 0xE002ED34 | MMFAR | MemManage Fault Address Register (NS) |
| 0xE002ED38 | BFAR | BusFault Address Register (NS) |
| 0xE002ED3C | AFSR | Auxiliary Fault Status Register (NS) |
| 0xE002ED40 | ID_PFR0 | Processor Feature Register 0 (NS) |
| 0xE002ED44 | ID_PFR1 | Processor Feature Register 1 (NS) |
| 0xE002ED48 | ID_DFR0 | Debug Feature Register 0 (NS) |
| 0xE002ED4C | ID_AFR0 | Auxiliary Feature Register 0 (NS) |
| 0xE002ED50 | ID_MMFR0 | Memory Model Feature Register 0 (NS) |
| 0xE002ED54 | ID_MMFR1 | Memory Model Feature Register 1 (NS) |
| 0xE002ED58 | ID_MMFR2 | Memory Model Feature Register 2 (NS) |
| 0xE002ED5C | ID_MMFR3 | Memory Model Feature Register 3 (NS) |
| 0xE002ED60 | ID_ISAR0 | Instruction Set Attribute Register 0 (NS) |
| 0xE002ED64 | ID_ISAR1 | Instruction Set Attribute Register 1 (NS) |
| 0xE002ED68 | ID_ISAR2 | Instruction Set Attribute Register 2 (NS) |
| 0xE002ED6C | ID_ISAR3 | Instruction Set Attribute Register 3 (NS) |
| 0xE002ED70 | ID_ISAR4 | Instruction Set Attribute Register 4 (NS) |
| 0xE002ED74 | ID_ISAR5 | Instruction Set Attribute Register 5 (NS) |
| 0xE002ED78 | CLIDR | Cache Level ID Register (NS) |
| 0xE002ED7C | CTR | Cache Type Register (NS) |
| 0xE002ED80 | CCSIDR | Current Cache Size ID register (NS) |
| 0xE002ED84 | CSSELR | Cache Size Selection Register (NS) |
| 0xE002ED88 | CPACR | Coprocessor Access Control Register (NS) |

### D1.1.24 Memory Protection Unit (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002ED90 | MPU_TYPE | MPU Type Register (NS) |
| 0xE002ED94 | MPU_CTRL | MPU Control Register (NS) |
| 0xE002ED98 | MPU_RNR | MPU Region Number Register (NS) |
| 0xE002ED9C | MPU_RBAR | MPU Region Base Address Register (NS) |
| 0xE002EDA0 | MPU_RLAR | MPU Region Limit Address Register (NS) |

| Address | Register | Description |
|---|---|---|
| 0xE002EDA4 | MPU_RBAR_An | MPU Region Base Address Register Alias *n* (NS) |
| 0xE002EDA8 | MPU_RLAR_An | MPU Region Limit Address Register Alias *n* (NS) |
| 0xE002EDC0 | MPU_MAIR0 | MPU Memory Attribute Indirection Register 0 (NS) |
| 0xE002EDC4 | MPU_MAIR1 | MPU Memory Attribute Indirection Register 1 (NS) |

### D1.1.25 Debug Control Block (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EDF0 | DHCSR | Debug Halting Control and Status Register (NS) |
| 0xE002EDF8 | DCRDR | Debug Core Register Data Register (NS) |
| 0xE002EDFC | DEMCR | Debug Exception and Monitor Control Register (NS) |
| 0xE002EE00 | DSCEMCR | Debug Set Clear Exception and Monitor Control Register (NS) |
| 0xE002EE04 | DAUTHCTRL | Debug Authentication Control Register (NS) |

### D1.1.26 Software Interrupt Generation (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EF00 | STIR | Software Triggered Interrupt Register (NS) |

### D1.1.27 Reliability, Availability and Serviceability Extension Fault Status Register (NS Alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EF04 | RFSR | RAS Fault Status Register (NS) |

### D1.1.28 Floating-Point Extension (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EF34 | FPCCR | Floating-Point Context Control Register (NS) |
| 0xE002EF38 | FPCAR | Floating-Point Context Address Register (NS) |
| 0xE002EF3C | FPDSCR | Floating-Point Default Status Control Register (NS) |
| 0xE002EF40 | MVFR0 | Media and VFP Feature Register 0 (NS) |
| 0xE002EF44 | MVFR1 | Media and VFP Feature Register 1 (NS) |
| 0xE002EF48 | MVFR2 | Media and VFP Feature Register 2 (NS) |

### D1.1.29 Cache Maintenance Operations (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EF50 | ICIALLU | Instruction Cache Invalidate All to PoU (NS) |
| 0xE002EF58 | ICIMVAU | Instruction Cache line Invalidate by Address to PoU (NS) |
| 0xE002EF5C | DCIMVAC | Data Cache line Invalidate by Address to PoC (NS) |
| 0xE002EF60 | DCISW | Data Cache line Invalidate by Set/Way (NS) |

| Address | Register | Description |
|---|---|---|
| 0xE002EF64 | DCCMVAU | Data Cache line Clean by address to PoU (NS) |
| 0xE002EF68 | DCCMVAC | Data Cache line Clean by Address to PoC (NS) |
| 0xE002EF6C | DCCSW | Data Cache Clean line by Set/Way (NS) |
| 0xE002EF70 | DCCIMVAC | Data Cache line Clean and Invalidate by Address to PoC (NS) |
| 0xE002EF74 | DCCISW | Data Cache line Clean and Invalidate by Set/Way (NS) |
| 0xE002EF78 | BPIALL | Branch Predictor Invalidate All (NS) |

### D1.1.30 Debug Identification Block (NS alias)

| Address | Register | Description |
|---|---|---|
| 0xE002EFB0 | DLAR | SCS Software Lock Access Register (NS) |
| 0xE002EFB4 | DLSR | SCS Software Lock Status Register (NS) |
| 0xE002EFB8 | DAUTHSTATUS | Debug Authentication Status Register (NS) |
| 0xE002EFBC | DDEVARCH | SCS Device Architecture Register (NS) |
| 0xE002EFCC | DDEVTYPE | SCS Device Type Register (NS) |
| 0xE002EFD0 | DPIDR4 | SCS Peripheral Identification Register 4 (NS) |
| 0xE002EFD4 | DPIDR5 | SCS Peripheral Identification Register 5 (NS) |
| 0xE002EFD8 | DPIDR6 | SCS Peripheral Identification Register 6 (NS) |
| 0xE002EFDC | DPIDR7 | SCS Peripheral Identification Register 7 (NS) |
| 0xE002EFE0 | DPIDR0 | SCS Peripheral Identification Register 0 (NS) |
| 0xE002EFE4 | DPIDR1 | SCS Peripheral Identification Register 1 (NS) |
| 0xE002EFE8 | DPIDR2 | SCS Peripheral Identification Register 2 (NS) |
| 0xE002EFEC | DPIDR3 | SCS Peripheral Identification Register 3 (NS) |
| 0xE002EFF0 | DCIDR0 | SCS Component Identification Register 0 (NS) |
| 0xE002EFF4 | DCIDR1 | SCS Component Identification Register 1 (NS) |
| 0xE002EFF8 | DCIDR2 | SCS Component Identification Register 2 (NS) |
| 0xE002EFFC | DCIDR3 | SCS Component Identification Register 3 (NS) |

### D1.1.31 Trace Port Interface Unit

| Address | Register | Description |
|---|---|---|
| 0xE0040000 | TPIU_SSPSR | TPIU Supported Parallel Port Sizes Register |
| 0xE0040004 | TPIU_CSPSR | TPIU Current Parallel Port Sizes Register |
| 0xE0040010 | TPIU_ACPR | TPIU Asynchronous Clock Prescaler Register |
| 0xE00400F0 | TPIU_SPPR | TPIU Selected Pin Protocol Register |
| 0xE0040300 | TPIU_FFSR | TPIU Formatter and Flush Status Register |
| 0xE0040304 | TPIU_FFCR | TPIU Formatter and Flush Control Register |
| 0xE0040308 | TPIU_PSCR | TPIU Periodic Synchronization Control Register |
| 0xE0040FB0 | TPIU_LAR | TPIU Software Lock Access Register |
| 0xE0040FB4 | TPIU_LSR | TPIU Software Lock Status Register |
| 0xE0040FC8 | TPIU_TYPE | TPIU Device Identifier Register |
| 0xE0040FCC | TPIU_DEVTYPE | TPIU Device Type Register |
| 0xE0040FD0 | TPIU_PIDR4 | TPIU Peripheral Identification Register 4 |
| 0xE0040FD4 | TPIU_PIDR5 | TPIU Peripheral Identification Register 5 |
| 0xE0040FD8 | TPIU_PIDR6 | TPIU Peripheral Identification Register 6 |
| 0xE0040FDC | TPIU_PIDR7 | TPIU Peripheral Identification Register 7 |
| 0xE0040FE0 | TPIU_PIDR0 | TPIU Peripheral Identification Register 0 |
| 0xE0040FE4 | TPIU_PIDR1 | TPIU Peripheral Identification Register 1 |

| Address | Register | Description |
|---|---|---|
| 0xE0040FE8 | TPIU_PIDR2 | TPIU Peripheral Identification Register 2 |
| 0xE0040FEC | TPIU_PIDR3 | TPIU Peripheral Identification Register 3 |
| 0xE0040FF0 | TPIU_CIDR0 | TPIU Component Identification Register 0 |
| 0xE0040FF4 | TPIU_CIDR1 | TPIU Component Identification Register 1 |
| 0xE0040FF8 | TPIU_CIDR2 | TPIU Component Identification Register 2 |
| 0xE0040FFC | TPIU_CIDR3 | TPIU Component Identification Register 3 |

## D1.2 Alphabetical list of registers

### D1.2.1 ACTLR, Auxiliary Control Register

The ACTLR characteristics are:

**Purpose**
Provides IMPLEMENTATION DEFINED configuration and control options.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000E008`.

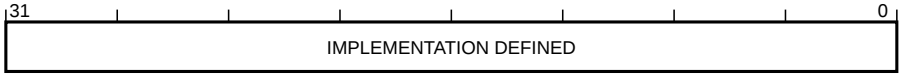
Secure software can access the Non-secure version of this register via ACTLR_NS located at `0xE002E008`. The location `0xE002E008` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The ACTLR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**
IMPLEMENTATION DEFINED. The contents of this field are IMPLEMENTATION DEFINED.

### D1.2.2 AFSR, Auxiliary Fault Status Register

The AFSR characteristics are:

**Purpose**
Provides IMPLEMENTATION DEFINED fault status information.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED3C`.

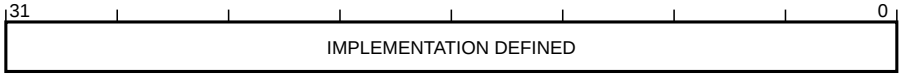
Secure software can access the Non-secure version of this register via AFSR_NS located at `0xE002ED3C`. The location `0xE002ED3C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The AFSR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**
IMPLEMENTATION DEFINED. The contents of this field are IMPLEMENTATION DEFINED.

### D1.2.3 AIRCR, Application Interrupt and Reset Control Register

The AIRCR characteristics are:

**Purpose**
Sets or returns interrupt control and reset configuration.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED0C`.

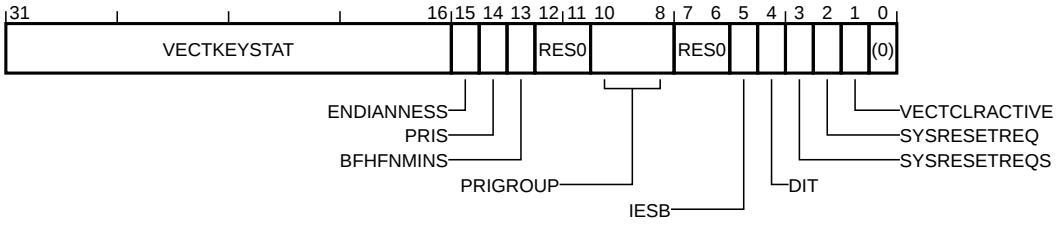
Secure software can access the Non-secure version of this register via AIRCR_NS located at `0xE002ED0C`. The location `0xE002ED0C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

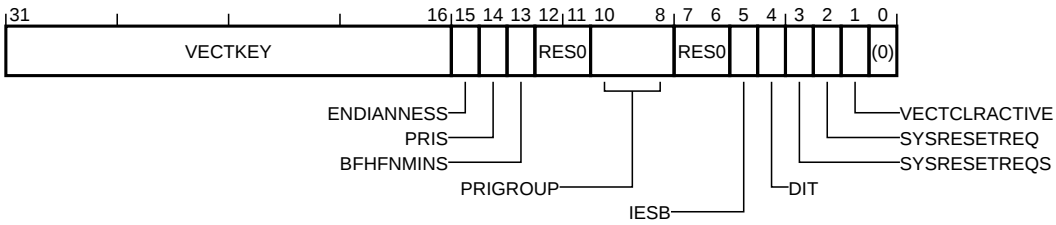
## Field descriptions

The AIRCR bit assignments are:

On a read:




On a write:




**VECTKEY, bits [31:16], on a write**
Vector key. Writes to the AIRCR must be accompanied by a write of the value `0x05FA` to this field. Writes to the AIRCR fields that are not accompanied by this value are ignored for the purpose of updating any of the AIRCR values or initiating any AIRCR functionality.

This field is not banked between Security states.

The possible values of this field are:

**`0x05FA`**
Permit write to AIRCR fields.

**Not `0x05FA`**
Accompanying write to AIRCR fields ignored.

**VECTKEYSTAT, bits [31:16], on a read**
Vector key status. Returns the bitwise inverse of the value required to be written to VECTKEY.

This field is not banked between Security states.

This field reads as `0xFA05`.

**ENDIANNESS, bit [15]**
Data endianness. Indicates how the PE interprets the memory system data endianness.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Little-endian.

**1**
Big-endian.

This bit is read-only.

This bit reads as an IMPLEMENTATION DEFINED value.

**PRIS, bit [14]**
Prioritize Secure exceptions. The value of this bit defines whether Secure exception priority boosting is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Priority ranges of Secure and Non-secure exceptions are identical.

**1**
Non-secure exceptions are de-prioritized.

To allow lock down of this bit, it is IMPLEMENTATION DEFINED whether this bit is writable.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**BFHFNMINS, bit [13]**
BusFault, HardFault, and NMI Non-secure enable. The value of this bit defines whether BusFault and NMI exceptions are Non-secure, and whether exceptions target the Non-secure HardFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
BusFault, HardFault, and NMI are Secure.

**1**
BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault.

If an implementation resets into Secure state, this bit resets to zero. If an implementation does not support Secure state, this bit is RAO/WI. To allow lock down of this field it is IMPLEMENTATION DEFINED whether this bit is writable. The effect of setting both BFHFNMINS and PRIS to 1 is UNPREDICTABLE.

This bit is read-only from Non-secure state.

This bit resets to zero on a Warm reset.

**Bits [12:11]**
Reserved, RES0.

**PRIGROUP, bits [10:8]**
Priority grouping. The value of this field defines the exception priority binary point position for the selected Security state.

This field is banked between Security states.

The possible values of this field are:

**0b000**
Group priority [7:1], subpriority [0].

**0b001**
Group priority [7:2], subpriority [1:0].

**0b010**
Group priority [7:3], subpriority [2:0].

**0b011**
Group priority [7:4], subpriority [3:0].

**0b100**
Group priority [7:5], subpriority [4:0].

**0b101**
Group priority [7:6], subpriority [5:0].

**0b110**
Group priority [7], subpriority [6:0].

**0b111**
No group priority, subpriority [7:0].

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

**Bits [7:6]**
Reserved, RES0.

**IESB, bit [5]**
Implicit ESB Enable. This bit indicates and allows modification of whether an implicit Error Synchronization Barriers occurs around lazy state preservation, and on every exception entry and return.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No Implicit ESB.

**1**
Implicit ESB are enabled.

Enabling implicit ESB's also causes imprecise BusFault exceptions to escalate as if they were precise, however because the address of the instruction that caused the fault is not known they are still reported as an IMPRECISERR in BFSR.

If a PE does not allow configuring implicit ESB insertion, this bit is WI and the value read indicates whether the PE never or always inserts an implicit ESB around lazy state preservation, and on exception entry and return.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If RAS is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**DIT, bit [4]**
Data Independent Timing. This bit indicates and allows modification of whether for the selected Security state data independent timing operations are guaranteed to be timing invariant with respect to the data values being operated on.

This bit is banked between Security states.

The possible values of this bit are:

**0**
The architecture makes no statement about the timing properties of any instructions.

**1**
Operations which the architecture defines as having data independent timing exhibit this behavior.

If configuration of timing invariance is not supported this bit is WI, and the value read indicates whether the PE always or never exhibits timing invariant behavior.

This bit resets to zero on a Warm reset.

**SYSRESETREQS, bit [3]**
System reset request Secure only. The value of this bit defines whether the SYSRESETREQ bit is functional for Non-secure use.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SYSRESETREQ functionality is available to both Security states.

**1**
SYSRESETREQ functionality is only available to Secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SYSRESETREQ, bit [2]**
System reset request. This bit allows software or a debugger to request a system reset.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Do not request a system reset.

**1**
Request a system reset.

When SYSRESETREQS is set to 1, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**VECTCLRACTIVE, bit [1]**
Clear active state.

A debugger write of one to this bit when the PE is halted in Debug state:

- IPSR is cleared to zero.
- If DHCSR.S_NSUIDE==0, the active state for all Non-secure exceptions is cleared.

- If DHCSR.S_SUIDE==0 and DHCSR.S_SDE==1, the active state for all Secure exceptions is cleared.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Do not clear active state.

**1**
Clear active state.

Writes to this bit while the PE is in Non-debug state are ignored.

This bit reads as zero.

**Bit [0]**
Reserved, RES0.

### D1.2.4 APSR, Application Program Status Register

The APSR characteristics are:

**Purpose**
Provides privileged and unprivileged access to the PE Execution state fields.

**Usage constraints**
Privileged and unprivileged access permitted.

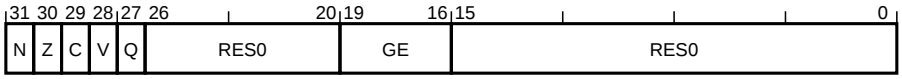
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The APSR bit assignments are:




**N, bit [31]**
Negative condition flag. When updated by a flag setting instruction this bit indicates whether the result of the operation when treated as a two's complement signed integer is negative.

The possible values of this bit are:

**0**
Result is positive or zero.

**1**
Result is negative.

See individual instruction pages for details.

This bit resets to an UNKNOWN value on a Warm reset.

**Z, bit [30]**
Zero condition flag. When updated by a flag setting instruction this bit indicates whether the result of the operation was zero.

The possible values of this bit are:

**0**
Result is non-zero.

**1**
Result is zero.

See individual instruction pages for details.

This bit resets to an UNKNOWN value on a Warm reset.

**C, bit [29]**
Carry condition flag. When updated by a flag setting instruction this bit indicates whether the operation resulted in an unsigned overflow or whether the last bit shifted out of the result was set.

The possible values of this bit are:

**0**
No carry occurred, or last bit shifted was clear.

**1**
Carry occurred, or last bit shifted was set.

See individual instruction pages for details.

This bit resets to an UNKNOWN value on a Warm reset.

**V, bit [28]**
Overflow condition flag. When updated by a flag setting instruction this bit indicates whether a signed overflow occurred.

The possible values of this bit are:

**0**
Signed overflow did not occur.

**1**
Signed overflow occurred.

See individual instruction pages for details.

This bit resets to an UNKNOWN value on a Warm reset.

**Q, bit [27]**
Sticky saturation flag. When updated by certain instructions this bit indicates either that an overflow occurred or that the result was saturated. This bit is cumulative and can only be cleared to zero by software.

The possible values of this bit are:

**0**
Saturation or overflow has not occurred since bit was last cleared.

**1**
Saturation or overflow has occurred since bit was last cleared.

See individual instruction pages for details.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [26:20]**
Reserved, RES0.

**GE, bits [19:16]**
Greater than or equal flags. When updated by parallel addition and subtraction instructions these bits record whether the result was greater than or equal to zero. SEL instructions use these bits to determine which register to select a particular byte from.

See individual instruction pages for details.

If the DSP Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

**Bits [15:0]**
Reserved, RES0.

### D1.2.5 BASEPRI, Base Priority Mask Register

The BASEPRI characteristics are:

**Purpose**
Changes the priority level required for exception preemption.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**
Present only if the Main Extension is implemented.

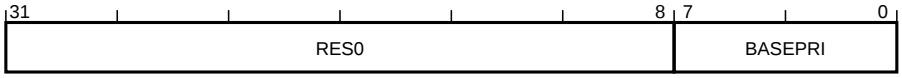
This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The BASEPRI bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**BASEPRI, bits [7:0]**
Base priority mask. BASEPRI changes the priority level required for exception preemption. It has an effect only when BASEPRI has a lower value than the unmasked priority level of the currently executing software.

The possible values of this field are:

**0**
Disables masking by BASEPRI.

**1-255**
Priority value.

The number of implemented bits in BASEPRI is the same as the number of implemented bits in each field of the priority registers, and BASEPRI has the same format as those fields.

This field resets to zero on a Warm reset.

### D1.2.6 BFAR, BusFault Address Register

The BFAR characteristics are:

**Purpose**
Shows the address associated with a precise data access BusFault.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000ED38`.

Secure software can access the Non-secure version of this register via BFAR_NS located at `0xE002ED38`. The location `0xE002ED38` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

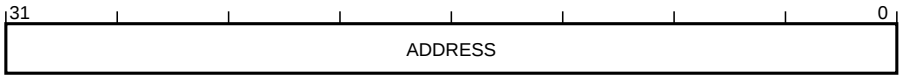
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

#### Preface

The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is set to 0.

#### Field descriptions

The BFAR bit assignments are:




**ADDRESS, bits [31:0]**
Data address for a precise BusFault. This register is updated with the address of a location that produced a BusFault. BFSR shows the reason for the fault. This field is valid only when BFSR.BFARVALID is set, otherwise it is UNKNOWN.

In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if MMFSR.MMARVALID is set.

If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

This field resets to an UNKNOWN value on a Warm reset.

**Note**

If an implementation shares a common BFAR and MMFAR it must not leak Secure state information to the Non-secure state. One possible implementation is that BFAR shares resource with the Secure MMFAR if AIRCR.BFHFNMINS is zero, and with the Non-secure MMFAR if AIRCR.BFHFNMINS is set.

### D1.2.7 BFSR, BusFault Status Register

The BFSR characteristics are:

**Purpose**

Shows the status of bus errors resulting from instruction fetches and data accesses. This includes errors resulting from RAS errors.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

8-bit read/write-one-to-clear register located at `0xE000ED29`.

Secure software can access the Non-secure version of this register via BFSR_NS located at `0xE002ED29`. The location `0xE002ED29` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

This register is part of CFSR.

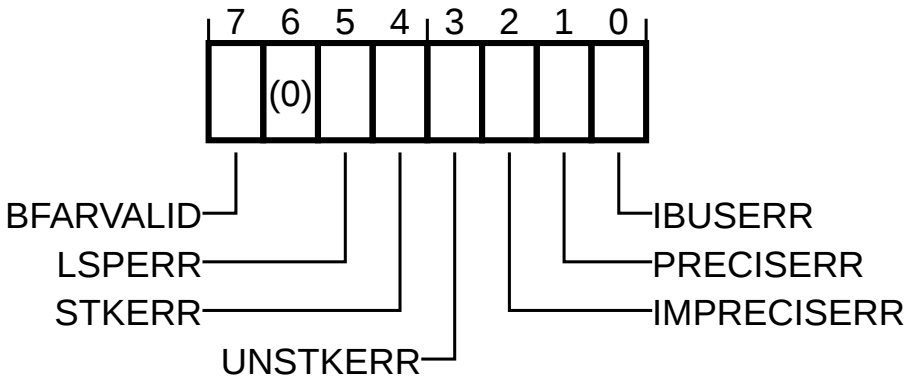
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

#### Preface

The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is set to 0.

#### Field descriptions

The BFSR bit assignments are:




**BFARVALID, bit [7]**

BFAR valid. Indicates validity of the contents of the BFAR register.

The possible values of this bit are:

**0**
BFAR content not valid.

**1**
BFAR content valid.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**Bit [6]**
Reserved, RES0.

**LSPERR, bit [5]**
Lazy state preservation error. Records whether a BusFault occurred during FP lazy state preservation.

The possible values of this bit are:

**0**
No BusFault occurred.

**1**
BusFault occurred.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**STKERR, bit [4]**
Stack error. Records whether a derived BusFault occurred during exception entry stacking.

The possible values of this bit are:

**0**
No derived BusFault occurred.

**1**
Derived BusFault occurred during exception entry.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**UNSTKERR, bit [3]**
Unstack error. Records whether a derived BusFault occurred during exception return unstacking.

The possible values of this bit are:

**0**
No derived BusFault occurred.

**1**
Derived BusFault occurred during exception return.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**IMPRECISERR, bit [2]**
Imprecise error. Records whether an imprecise data access error has occurred.

The possible values of this bit are:

**0**
No imprecise data access error has occurred.

**1**
Imprecise data access error has occurred.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**PRECISERR, bit [1]**
Precise error. Records whether a precise data access error has occurred.

The possible values of this bit are:

**0**
No precise data access error has occurred.

**1**
Precise data access error has occurred.

When a precise error is recorded, the associated address is written to the BFAR and BFSR.BFARVALID bit is set.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**IBUSERR, bit [0]**
Instruction bus error. Records whether a BusFault on an instruction prefetch has occurred.

The possible values of this bit are:

**0**
No BusFault on instruction prefetch has occurred.

**1**
A BusFault on an instruction prefetch has occurred.

An IBUSERR is only recorded if the instruction is issued for execution.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

### D1.2.8 BPIALL, Branch Predictor Invalidate All

The BPIALL characteristics are:

**Purpose**
Invalidate all entries from branch predictors.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

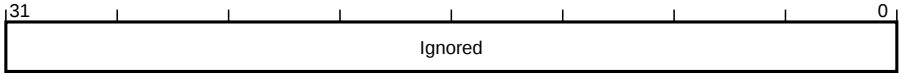
**Attributes**
32-bit write-only register located at `0xE000EF78`.

Secure software can access the Non-secure version of this register via BPIALL_NS located at `0xE002EF78`. The location `0xE002EF78` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The BPIALL bit assignments are:




**Ignored, bits [31:0]**
Ignored. The value written to this field is ignored.

### D1.2.9 CCR, Configuration and Control Register

The CCR characteristics are:

**Purpose**
Sets or returns configuration and control data.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible for read accesses through unprivileged DAP requests when DAUTHCTRL.UIDAPEN (either bank) is set.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED14`.

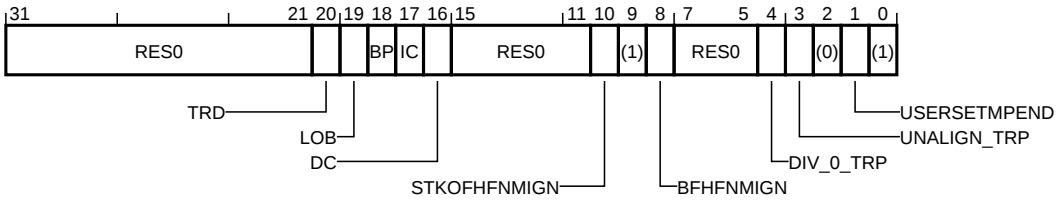
Secure software can access the Non-secure version of this register via CCR_NS located at `0xE002ED14`. The location `0xE002ED14` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The CCR bit assignments are:




**Bits [31:21]**
Reserved, RES0.

**TRD, bit [20]**
Thread reentrancy disabled. Enables checking for exception stack frame integrity signatures on SG instructions. If enabled this check causes a fault to be raised if an attempt is made to re-enter Secure Thread mode while a call to Secure Thread mode is already in progress.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Integrity signature checking not performed.

**1**
Integrity signature checking performed.

This bit is RAZ/WI from Non-secure state.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**LOB, bit [19]**
Loop and branch info cache enable. Enables the branch cache used by loop and branch future instructions for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Branch cache disabled for the selected Security state.

**1**
Branch cache enabled for the selected Security state.

If the branch cache is not supported, this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**BP, bit [18]**
Branch prediction enable. Enables program flow prediction for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Program flow prediction disabled for the selected Security state.

**1**
Program flow prediction enabled for the selected Security state.

If program flow prediction cannot be disabled, this bit is RAO/WI. If the program flow prediction is not supported, this bit is RAZ/WI.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**IC, bit [17]**
Instruction cache enable. This is a global enable bit for instruction caches in the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Instruction caches disabled for the selected Security state.

**1**
Instruction caches enabled for the selected Security state.

If the PE does not implement instruction caches, this bit is RAZ/WI.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**DC, bit [16]**
Data cache enable. Enables data caching of all data accesses to Normal memory.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Data caching disabled.

**1**
Data caching enabled.

The secure version of this bit controls the Cacheability of accesses to secure memory.

The non-secure version of this bit controls the Cacheability of accesses to non-secure memory.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bits [15:11]**
Reserved, RES0.

**STKOFHFNMIGN, bit [10]**
Stack overflow in HardFault and NMI ignore. Controls the effect of a stack limit violation while executing at a requested priority less than 0 for the Security state with which the stack limit register is associated.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Stack limit faults not ignored.

**1**
Stack limit faults at requested priorities of less than 0 ignored.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bit [9]**
Reserved, RES1.

**BFHFNMIGN, bit [8]**
BusFault in HardFault or NMI ignore. Determines the effect of precise BusFaults on handlers running at a requested priority less than 0.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Precise BusFaults not ignored.

**1**
Precise BusFaults at requested priorities of less than 0 ignored.

If AIRCR.BFHFNMINS is 0, this bit is read-only from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bits [7:5]**
Reserved, RES0.

**DIV_0_TRP, bit [4]**
Divide by zero trap. Controls the generation of a DIVBYZERO UsageFault when attempting to perform integer division by zero.

This bit is banked between Security states.

The possible values of this bit are:

**0**
DIVBYZERO UsageFault generation disabled.

**1**
DIVBYZERO UsageFault generation enabled.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**UNALIGN_TRP, bit [3]**
Unaligned trap. Controls the trapping of unaligned word or halfword accesses.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Unaligned accesses permitted from LDR, LDRH, STR, and STRH.

**1**
Any unaligned transaction generates an UNALIGNED UsageFault.

Unaligned load/store multiples and atomic/exclusive accesses always generate an UNALIGNED UsageFault.

If the Main Extension is not implemented, this bit is RES1.

This bit resets to zero on a Warm reset if the Main Extension is implemented.

**Bit [2]**
Reserved, RES0.

**USERSETMPEND, bit [1]**
User set main pending. Determines whether unprivileged accesses are permitted to pend interrupts via the STIR.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Unprivileged accesses to the STIR generate a fault.

**1**
Unprivileged accesses to the STIR are permitted.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bit [0]**
Reserved, RES1.

### D1.2.10 CCSIDR, Current Cache Size ID register

The CCSIDR characteristics are:

**Purpose**
The CCSIDR provides information about the architecture of the currently selected cache.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If CSSELR points to an unimplemented cache, the value of this register is UNKNOWN.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read-only register located at `0xE000ED80`.

Secure software can access the Non-secure version of this register via CCSIDR_NS located at `0xE002ED80`. The location `0xE002ED80` is RES0 to software executing in Non-secure state and the debugger.

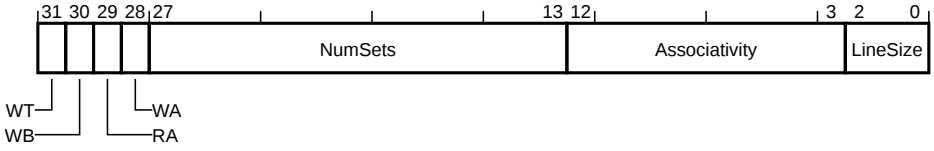
This register is banked between Security states.

## Preface

Provides indirect read access to the architecture of the cache currently selected by CSSELR. The parameters NumSets, Associativity, and LineSize in these registers define the architecturally visible parameters that are required for the cache maintenance by Set/Way instructions. They are not guaranteed to represent the actual microarchitectural features of a design. You cannot make any inference about the actual sizes of caches based on these parameters.

## Field descriptions

The CCSIDR bit assignments are:




**WT, bit [31]**
Write-Through. Indicates whether the currently selected cache level supports Write-Through.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**WB, bit [30]**
Writeback. Indicates whether the currently selected cache level supports Write-Back.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**RA, bit [29]**
Read-allocate. Indicates whether the currently selected cache level supports read-allocation.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**WA, bit [28]**
Write-Allocate. Indicates whether the currently selected cache level supports write-allocation.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**NumSets, bits [27:13]**
Number of sets. Indicates (Number of sets in the currently selected cache) - 1. Therefore, a value of 0 indicates that 1 is set in the cache. The number of sets does not have to be a power of 2.

This field reads as an IMPLEMENTATION DEFINED value.

**Associativity, bits [12:3]**
Associativity. Indicates (Associativity of cache) - 1. A value of 0 indicates an associativity of 1. The associativity does not have to be a power of 2.

This field reads as an IMPLEMENTATION DEFINED value.

**LineSize, bits [2:0]**
Line size. Indicates ($Log_2$(Number of words per line in the currently selected cache)) - 2.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.11 CFSR, Configurable Fault Status Register

The CFSR characteristics are:

**Purpose**
Contains the three Configurable Fault Status Registers.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write-one-to-clear register located at `0xE000ED28`.

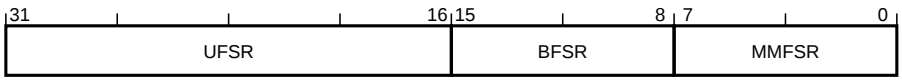
Secure software can access the Non-secure version of this register via CFSR_NS located at `0xE002ED28`. The location `0xE002ED28` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The CFSR bit assignments are:




**UFSR, bits [31:16]**
UsageFault Status Register. Provides information on UsageFault exceptions.

This field is banked between Security states.

See UFSR.

This field resets to zero on a Warm reset.

**BFSR, bits [15:8]**
BusFault Status Register. Provides information on BusFault exceptions.

This field is not banked between Security states.

See BFSR.

This field resets to zero on a Warm reset.

**MMFSR, bits [7:0]**
MemManage Fault Status Register. Provides information on MemManage exceptions.

This field is banked between Security states.

See MMFSR.

This field resets to zero on a Warm reset.

### D1.2.12 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

**Purpose**
Identifies the type of caches implemented and the level of coherency and unification.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

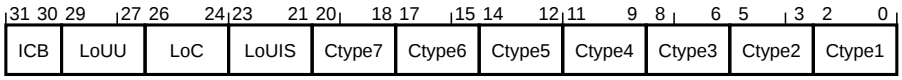
**Attributes**
32-bit read-only register located at `0xE000ED78`.

Secure software can access the Non-secure version of this register via CLIDR_NS located at `0xE002ED78`. The location `0xE002ED78` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The CLIDR bit assignments are:




**ICB, bits [31:30]**
Inner cache boundary. This field indicates the boundary between inner and outer domain.

The possible values of this field are:

**`0b00`**
Not disclosed in this mechanism.

**`0b01`**
L1 cache is the highest inner level.

**`0b10`**
L2 cache is the highest inner level.

**`0b11`**
L3 cache is the highest inner level.

This field reads as an IMPLEMENTATION DEFINED value.

**LoUU, bits [29:27]**
Level of Unification Uniprocessor. This field indicates the Level of Unification Uniprocessor for the cache hierarchy.

This field reads as an IMPLEMENTATION DEFINED value.

**LoC, bits [26:24]**
Level of Coherence. This field indicates the Level of Coherence for the cache hierarchy.

This field reads as an IMPLEMENTATION DEFINED value.

**LoUIS, bits [23:21]**
Level of Unification Inner Shareable. This field indicates the Level of Unification Shareable for the cache hierarchy.

This field reads as an IMPLEMENTATION DEFINED value.

**Ctype*m*, bits [3(*m*-1)+2:3(*m*-1)], for *m* = 1 to 7**
Cache type field *m*. Indicates the type of cache implemented at level *m*.

The possible values of this field are:

**0b000**
No cache.

**0b001**
Instruction cache only.

**0b010**
Data cache only.

**0b011**
Separate instruction and data caches.

**0b100**
Unified cache.

All other values are reserved.

If Ctype<*m*> is set to 0b000, and *m* < 7, then all of the following apply.

Level *m* represents the last level of software-visible cache.

Ctype<*m*=1> through to Ctype7 must read as zero.

Software must treat Ctype<*m*+1> through Ctype7 as if they are invalid and read as an UNKNOWN value.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.13 CONTROL, Control Register

The CONTROL characteristics are:

**Purpose**
Provides access to the PE control fields.

**Usage constraints**
Privileged access only, but unprivileged writes are ignored unless otherwise specified.

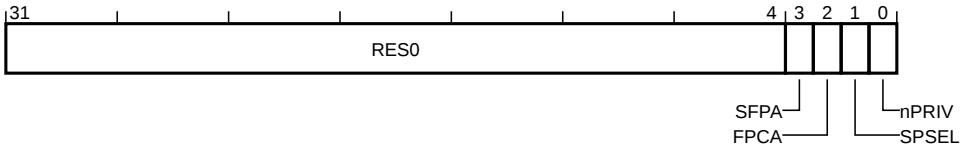
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The CONTROL bit assignments are:




**Bits [31:4]**
Reserved, RES0.

**SFPA, bit [3]**
Secure Floating-point active. Indicates that the Floating-point registers contain active state that belongs to the Secure state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
The Floating-point registers do not contain state that belongs to the Secure state.

**1**
The Floating-point registers contain state that belongs to the Secure state.

This bit is accessible from both privileged and unprivileged modes, but unprivileged writes are ignored.

This bit is RAZ/WI from Non-secure state.

If the Security Extension is not implemented, this bit is RES0.

If the Floating-point Extension and MVE are not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**FPCA, bit [2]**
Floating-point context active. Defines whether the FP Extension is active in the current context.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
FP Extension is not active.

**1**
FP Extension is active.

When NSACR.CP10 is set to zero, the Non-secure view of this bit is read-only. If FPCCR.ASPEN is set to 1, enabling automatic Floating-point state preservation, then the PE sets this bit to 1 on successful completion of any Floating-point instruction.

If the Floating-point Extension and MVE are not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**SPSEL, bit [1]**
Stack-pointer select. Defines the stack pointer to be used.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Use SP_main as the current stack.

**1**
In Thread mode use SP_process as the current stack.

This bit resets to zero on a Warm reset.

**nPRIV, bit [0]**
Not privileged. Defines the execution privilege in Thread mode.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Thread mode has privileged access.

**1**
Thread mode has unprivileged access only.

If the Main Extension is not implemented, it is IMPLEMENTATION DEFINED whether this field is RW or RAZ/WI.

This bit resets to zero on a Warm reset.

### D1.2.14 CPACR, Coprocessor Access Control Register

The CPACR characteristics are:

**Purpose**
Specifies the access privileges for coprocessors and the Floating-point Extension. If MVE is implemented, this register specifies the access privileges for MVE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000ED88`.

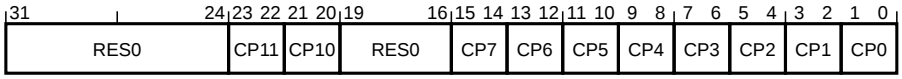
Secure software can access the Non-secure version of this register via CPACR_NS located at `0xE002ED88`. The location `0xE002ED88` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The CPACR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**CP11, bits [23:22]**
CP11 Privilege. The value in this field is ignored. If the CP10 field is RAZ/WI this field is also RAZ/WI. If the value of this field is not programmed to the same value as the CP10 field, then the value is UNKNOWN.

This field resets to an UNKNOWN value on a Warm reset.

**CP10, bits [21:20]**
CP10 Privilege. Defines the access rights for the Floating-point functionality.

The possible values of this field are:

**`0b00`**
All accesses to the FP Extension and MVE result in NOCP UsageFault.

**`0b01`**
Unprivileged accesses to the FP Extension and MVE result in NOCP UsageFault.

**`0b11`**
Full access to the FP Extension and MVE.

All other values are reserved.

The features controlled by this field are:

Unless otherwise specified, the execution of any instructions within the encoding space defined by IsCPInstruction() for CP10.

Access to any Floating-point registers in the range D0-D16.

See individual instruction pages for details.

If the Floating-point Extension and MVE are not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

**Bits [19:16]**
Reserved, RES0.

**CP*m*, bits [2*m*+1:2*m*], for *m* = 0 to 7**
Coprocessor *m* privilege. Controls access privileges for coprocessor *m*.

The possible values of this field are:

**0b00**
Access denied. Any attempted access generates a NOCP UsageFault.

**0b01**
Privileged access only. An unprivileged access generates a NOCP UsageFault.

**0b10**
Reserved.

**0b11**
Full access.

If coprocessor *m* is not implemented, this field is RAZ/WI.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.15 CPPWR, Coprocessor Power Control Register

The CPPWR characteristics are:

**Purpose**
Specifies whether coprocessors are permitted to enter a non-retentive power state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000E00C`.

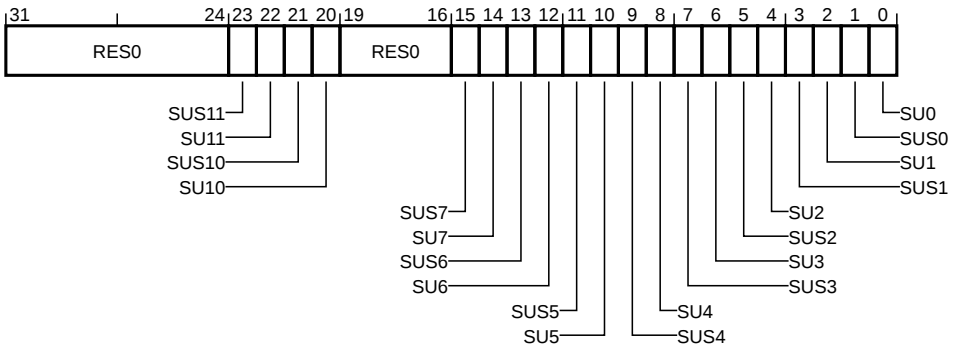
Secure software can access the Non-secure version of this register via CPPWR_NS located at `0xE002E00C`. The location `0xE002E00C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The CPPWR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**SUS11, bit [23]**
State UNKNOWN Secure only 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SUS10 field, then the value is UNKNOWN.

If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SU11, bit [22]**
State UNKNOWN 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SU10 field, then the value is UNKNOWN.

When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit resets to zero on a Warm reset.

**SUS10, bit [21]**
State UNKNOWN Secure only 10. This bit indicates and allows modification of whether the SU10 field can be modified from Non-secure state.

The possible values of this bit are:

**0**
The SU10 field is accessible from both Security states.

**1**
The SU10 field is only accessible from the Secure state.

If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SU10, bit [20]**
State UNKNOWN 10. This bit indicates and allows modification of whether the state associated with the Floating-point and MVE units is permitted to become UNKNOWN. This can be used as a hint to power control logic that these units might be powered down.

The possible values of this bit are:

**0**
The Floating-point and MVE state is not permitted to become UNKNOWN.

**1**
The Floating-point and MVE state is permitted to become UNKNOWN.

When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. It is IMPLEMENTATION DEFINED whether this bit is always RAZ/WI.

This bit resets to zero on a Warm reset.

**Bits [19:16]**
Reserved, RES0.

**SUS*m*, bit [2*m*+1], for *m* = 0 to 7**
State UNKNOWN Secure only *m*. This field indicates and allows modification of whether the SU*m* field can be modified from Non-secure state.

The possible values of this field are:

**0**
The SU*m* field is accessible from both Security states.

**1**
The SU*m* field is only accessible from the Secure state.

If SU*m* is always RAZ/WI this field is also RAZ/WI.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

**SU*m*, bit [2*m*], for *m* = 0 to 7**
State UNKNOWN *m*. This field indicates and allows modification of whether the state associated with coprocessor *m* is permitted to become UNKNOWN. This can be used as a hint to power control logic that the coprocessor might be powered down.

The possible values of this field are:

**0**
The coprocessor state is not permitted to become UNKNOWN.

**1**
The coprocessor state is permitted to become UNKNOWN.

When SUS*m* is set to 1, the Non-secure view of this bit is RAZ/WI. It is IMPLEMENTATION DEFINED whether this bit is always RAZ/WI.

This field resets to zero on a Warm reset.

### D1.2.16 CPUID, CPUID Base Register

The CPUID characteristics are:

**Purpose**
Provides identification information for the PE, including an implementer code for the device and a device ID number.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

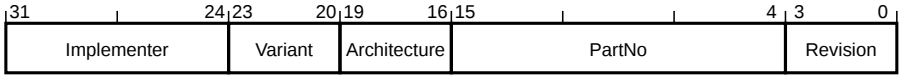
**Attributes**
32-bit read-only register located at `0xE000ED00`.

Secure software can access the Non-secure version of this register via CPUID_NS located at `0xE002ED00`. The location `0xE002ED00` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The CPUID bit assignments are:




**Implementer, bits [31:24]**
Implementer code. This field must hold an implementer code that has been assigned by Arm.

The possible values of this field are:

**`0x41`**
'A': Arm Limited.

**Not `0x41`**
Implementer other than Arm Limited.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field reads as an IMPLEMENTATION DEFINED value.

**Variant, bits [23:20]**
Variant number. IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field reads as an IMPLEMENTATION DEFINED value.

**Architecture, bits [19:16]**
Architecture version. Defines the Architecture implemented by the PE.

The possible values of this field are:

**`0b1100`**
Armv8-M architecture without Main Extension.

**0b1111**
Armv8-M architecture with Main Extension.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**PartNo, bits [15:4]**
Part number. IMPLEMENTATION DEFINED primary part number for the device.

This field reads as an IMPLEMENTATION DEFINED value.

**Revision, bits [3:0]**
Revision number. IMPLEMENTATION DEFINED revision number for the device.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.17 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

**Purpose**

Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type (either instruction or data cache)

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

This register is always implemented.

**Attributes**

32-bit read/write register located at `0xE000ED84`.

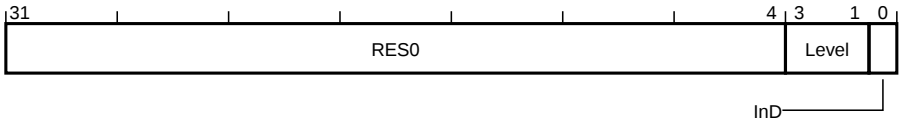
Secure software can access the Non-secure version of this register via CSSELR_NS located at `0xE002ED84`. The location `0xE002ED84` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The CSSELR bit assignments are:




**Bits [31:4]**

Reserved, RES0.

**Level, bits [3:1]**

Cache level. Selects which cache level is to be identified. Permitted values are from `0b000`, indicating Level 1 cache, to `0b110` indicating Level 7 cache.

The possible values of this field are:

**`0b000`**
Level 1 cache.

**`0b001`**
Level 2 cache.

**`0b010`**
Level 3 cache.

**`0b011`**
Level 4 cache.

**`0b100`**
Level 5 cache.

**0b101**
Level 6 cache.

**0b110**
Level 7 cache.

All other values are reserved.

Writing a reserved value or value corresponding to an unimplemented level of cache is *constrained unpredictable*.

This field resets to an UNKNOWN value on a Warm reset.

**InD, bit [0]**
Instruction not data. Selects whether the instruction or the data cache is to be identified.

The possible values of this bit are:

**0**
Data or unified cache.

**1**
Instruction cache.

This bit resets to an UNKNOWN value on a Warm reset.

## D1.2.18 CTR, Cache Type Register

The CTR characteristics are:

**Purpose**
Provides information about the architecture of the caches.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read-only register located at `0xE000ED7C`.

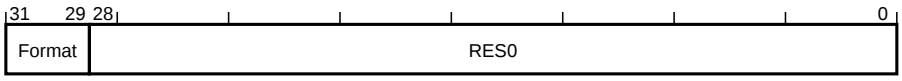
Secure software can access the Non-secure version of this register via CTR_NS located at `0xE002ED7C`. The location `0xE002ED7C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

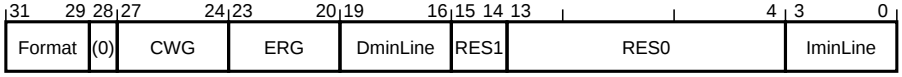
### Field descriptions

The CTR bit assignments are:

**When Format!='0b100'**:




**When Format=='0b100'**:




**Format, bits [31:29]**
Cache Type Register format. Indicates whether cache type information is provided.

The possible values of this field are:

**`0b000`**
No cache type information is provided.

**`0b100`**
Cache type information is provided.

All other values are reserved.

The value of this field is an IMPLEMENTATION DEFINED choice of either `0b000` or `0b100`.

If CLIDR is nonzero then this field must read as `0b100`.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [28:0], when Format!='0b100'**
Reserved, RES0.

**Bit [28], when Format==’0b100’**
Reserved, RES0.

**CWG, bits [27:24], when Format==’0b100’**
Cache Write-Back Granule. $Log_2$ of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

The possible values of this field are:

**0b0000**
Indicates that this register does not provide Cache Write-Back Granule information and either the architectural maximum of 512 words (2KB) must be assumed, or the Cache Write-Back Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

**0b0001-0b1000**
$Log_2$ of the number of words.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**ERG, bits [23:20], when Format==’0b100’**
Exclusives Reservation Granule. $Log_2$ of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The possible values of this field are:

**0b0000**
Indicates that this register does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2KB) must be assumed.

**0b0001-0b1000**
$Log_2$ of the number of words.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**DminLine, bits [19:16], when Format==’0b100’**
Data cache minimum line length. $Log_2$ of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [15:14], when Format==’0b100’**
Reserved, RES1.

**Bits [13:4], when Format==’0b100’**
Reserved, RES0.

**IminLine, bits [3:0], when Format==’0b100’**
Instruction cache minimum line length. $Log_2$ of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.19 DAUTHCTRL, Debug Authentication Control Register

The DAUTHCTRL characteristics are:

**Purpose**
This register allows the IMPLEMENTATION DEFINED authentication interface to be overridden from software.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If Armv8.1-M is implemented, this register is word, halfword and byte accessible.

This register is RES0 if accessed via the debugger.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

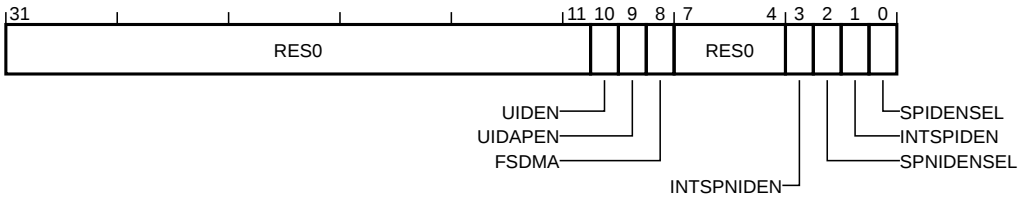
**Attributes**
32-bit read/write register located at `0xE000EE04`.

Secure software can access the Non-secure version of this register via DAUTHCTRL_NS located at `0xE002EE04`. The location `0xE002EE04` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

## Field descriptions

The DAUTHCTRL bit assignments are:




**Bits [31:11]**
Reserved, RES0.

**UIDEN, bit [10]**
Unprivileged Invasive Debug Enable. Enables halting debug for unprivileged modes, regardless of the state of other debug controls.

This bit is banked between Security states.

The possible values of this bit are:

**`0b0`**
Halting debug operates as normal.

**`0b1`**
Unprivileged halting debug allowed.

See UnprivHaltingDebugAllowed() and UpdateDebugEnable().

If UDE is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**UIDAPEN, bit [9]**
Unprivileged Invasive DAP Access Enable. Enables unprivileged DAP access to specific registers.

This bit is banked between Security states.

The possible values of this bit are:

**0b0**
No unprivileged DAP access.

**0b1**
Unprivileged DAP access allowed.

Unprivileged DAP access is allowed if either of the banked fields is set.

If UDE is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**FSDMA, bit [8]**
Force Secure DebugMonitor Allowed. Allows Secure DebugMonitor to be enabled without having to enable secure halting debug.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Secure DebugMonitor determined by other means.

**1**
Secure DebugMonitor allowed.

This bit is RAZ/WI from Non-secure state.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bits [7:4]**
Reserved, RES0.

**INTSPNIDEN, bit [3]**
Internal Secure non-invasive debug enable. Overrides the external Secure non-invasive debug authentication interface.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Secure Non-invasive debug prohibited.

**1**
Secure Non-invasive debug allowed.

Ignored if DAUTHCTRL.SPNIDENSEL == 0. See SecureNoninvasiveDebugAllowed().

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Cold reset.

**SPNIDENSEL, bit [2]**

Secure non-invasive debug enable select. Selects between DAUTHCTRL and the IMPLEMENTATION DEFINED external authentication interface for control of Secure non-invasive debug.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Secure non-invasive debug controlled by the IMPLEMENTATION DEFINED external authentication interface. In the CoreSight authentication interface, this is controlled by the SPNIDEN signal.

**1**

Secure non-invasive debug controlled by DAUTHCTRL.INTSPNIDEN.

The PE ignores the value of this bit and Secure non-invasive debug is allowed if DHCSR.S_SDE == 1. See SecureNoninvasiveDebugAllowed().

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Cold reset.

**INTSPIDEN, bit [1]**

Internal Secure invasive debug enable. Overrides the external Secure invasive debug authentication interfaces.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Secure halting and self-hosted debug prohibited.

**1**

Secure halting and self-hosted debug allowed.

Ignored if DAUTHCTRL.SPIDENSEL == 0. See SecureHaltingDebugAllowed() and SecureDebugMonitorAllowed().

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Cold reset.

**SPIDENSEL, bit [0]**

Secure invasive debug enable select. Selects between DAUTHCTRL and the IMPLEMENTATION DEFINED external authentication interface for control of Secure invasive debug.

This bit is not banked between Security states.

The possible values of this bit are:

**0**

Secure halting and self-hosted debug controlled by the IMPLEMENTATION DEFINED external authentication interface. In the CoreSight authentication interface, both are controlled by the SPIDEN signal.

**1**

Secure halting and self-hosted debug controlled by DAUTHCTRL.INTSPIDEN.

See SecureHaltingDebugAllowed() and SecureDebugMonitorAllowed().

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Cold reset.

## D1.2.20 DAUTHSTATUS, Debug Authentication Status Register

The DAUTHSTATUS characteristics are:

**Purpose**
Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for debug.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

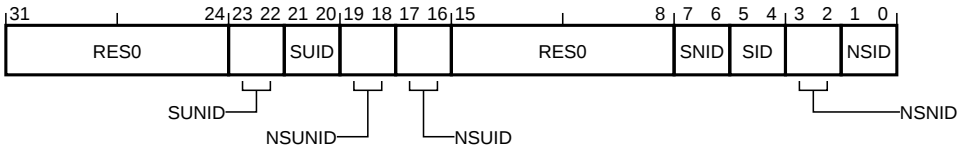
**Attributes**
32-bit read-only register located at `0xE000EFB8`.

Secure software can access the Non-secure version of this register via DAUTHSTATUS_NS located at `0xE002EFB8`. The location `0xE002EFB8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DAUTHSTATUS bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**SUNID, bits [23:22]**
Secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive debug is allowed for the Secure state.

The possible values of this field are:

**`0b00`**
Security Extension or Unprivileged Non-invasive Debug not implemented.

**`0b01`**
Reserved.

**`0b10`**
Secure Non-invasive debug prohibited.

**0b11**
Secure Non-invasive debug allowed in unprivileged mode.

If UDE is not implemented, this field is RES0.

**SUID, bits [21:20]**
Secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Halting Debug is allowed for the Secure state.

The possible values of this field are:

**0b00**
Security Extension or Unprivileged Debug not implemented.

**0b01**
Reserved.

**0b10**
Secure halting debug prohibited.

**0b11**
Secure halting debug allowed in unprivileged mode.

If UDE is not implemented, this field is RES0.

**NSUNID, bits [19:18]**
Non-secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive Debug is allowed for the Non-secure state.

The possible values of this field are:

**0b00**
Unprivileged Non-invasive debug not implemented.

**0b01**
Reserved.

**0b10**
Non-secure Non-invasive debug prohibited.

**0b11**
Non-secure Non-invasive debug allowed in unprivileged mode.

If UDE is not implemented, this field is RES0.

**NSUID, bits [17:16]**
Non-secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Halting Debug is allowed for the Non-secure state.

The possible values of this field are:

**0b00**
Unprivileged halting debug not implemented.

**0b01**
Reserved.

**0b10**
Non-secure halting debug prohibited.

**0b11**
Non-secure halting debug allowed in unprivileged mode.

If UDE is not implemented, this field is RES0.

**Bits [15:8]**
Reserved, RES0.

**SNID, bits [7:6]**
Secure Non-invasive Debug. Indicates whether Secure non-invasive debug is implemented and allowed.

The possible values of this field are:

**0b00**
Security Extension not implemented.

**0b01**
Reserved.

**0b10**
Security Extension implemented and Secure non-invasive debug prohibited.

**0b11**
Security Extension implemented and Secure non-invasive debug allowed.

**SID, bits [5:4]**
Secure Invasive Debug. Indicates whether Secure invasive debug is implemented and allowed.

The possible values of this field are:

**0b00**
Security Extension not implemented.

**0b01**
Reserved.

**0b10**
Security Extension implemented and Secure invasive debug prohibited.

**0b11**
Security Extension implemented and Secure invasive debug allowed.

**NSNID, bits [3:2]**
Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug is allowed.

The possible values of this field are:

**0b0x**
Reserved.

**0b10**
Non-secure non-invasive debug prohibited.

**0b11**
Non-secure non-invasive debug allowed.

**NSID, bits [1:0]**
Non-secure Invasive Debug. Indicates whether Non-secure invasive debug is allowed.

The possible values of this field are:

**0b0x**
Reserved.

**0b10**
Non-secure invasive debug prohibited.

**0b11**
Non-secure invasive debug allowed.

### D1.2.21 DCCIMVAC, Data Cache line Clean and Invalidate by Address to PoC

The DCCIMVAC characteristics are:

**Purpose**
Clean and invalidate data or unified cache line by address to PoC.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

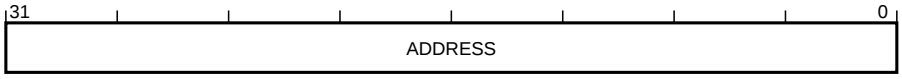
**Attributes**
32-bit write-only register located at `0xE000EF70`.

Secure software can access the Non-secure version of this register via DCCIMVAC_NS located at `0xE002EF70`. The location `0xE002EF70` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCCIMVAC bit assignments are:




**ADDRESS, bits [31:0]**
Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.22 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

**Purpose**
Clean and invalidate data or unified cache line by set/way.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

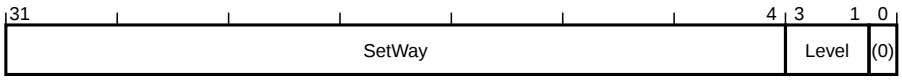
**Attributes**
32-bit write-only register located at `0xE000EF74`.

Secure software can access the Non-secure version of this register via DCCISW_NS located at `0xE002EF74`. The location `0xE002EF74` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCCISW bit assignments are:




**SetWay, bits [31:4]**
Cache set/way. Contains two fields: Way, bits[31:32-A], the number of the way to operate on. Set, bits[B-1:L], the number of the set to operate on. Bits[L-1:4] are RES0. A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS). ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**
Cache level. Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

### D1.2.23 DCCMVAC, Data Cache line Clean by Address to PoC

The DCCMVAC characteristics are:

**Purpose**
Clean data or unified cache line by address to PoC.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

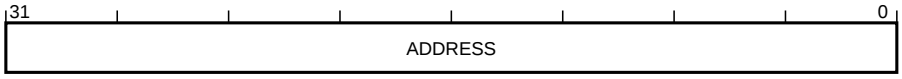
**Attributes**
32-bit write-only register located at `0xE000EF68`.

Secure software can access the Non-secure version of this register via DCCMVAC_NS located at `0xE002EF68`. The location `0xE002EF68` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCCMVAC bit assignments are:




**ADDRESS, bits [31:0]**
Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.24 DCCMVAU, Data Cache line Clean by address to PoU

The DCCMVAU characteristics are:

**Purpose**
Clean data or unified cache line by address to PoU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

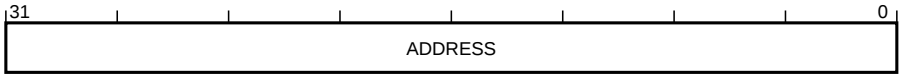
**Attributes**
32-bit write-only register located at `0xE000EF64`.

Secure software can access the Non-secure version of this register via DCCMVAU_NS located at `0xE002EF64`. The location `0xE002EF64` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCCMVAU bit assignments are:




**ADDRESS, bits [31:0]**
Address. Writing to this field initiates the maintenance operation for the address written.

## D1.2.25 DCCSW, Data Cache Clean line by Set/Way

The DCCSW characteristics are:

**Purpose**
Clean data or unified cache line by set/way.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

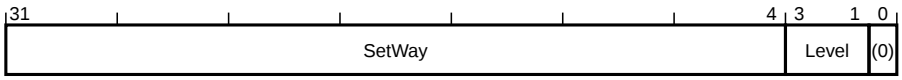
**Attributes**
32-bit write-only register located at `0xE000EF6C`.

Secure software can access the Non-secure version of this register via DCCSW_NS located at `0xE002EF6C`. The location `0xE002EF6C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCCSW bit assignments are:




**SetWay, bits [31:4]**
Cache set/way. Contains two fields: Way, bits [31:32-A], the number of the way to operate on. Set, bits [B-1:L], the number of the set to operate on. Bits [L-1:4] are RES0. A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS). ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**
Cache level. Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

## D1.2.26 DCIDR0, SCS Component Identification Register 0

The DCIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

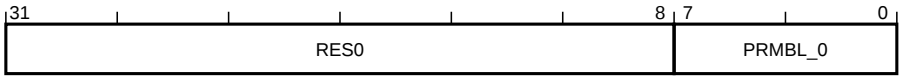
**Attributes**
32-bit read-only register located at `0xE000EFF0`.

Secure software can access the Non-secure version of this register via DCIDR0_NS located at `0xE002EFF0`. The location `0xE002EFF0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_0, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

## D1.2.27 DCIDR1, SCS Component Identification Register 1

The DCIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

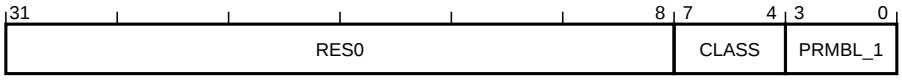
**Attributes**
32-bit read-only register located at `0xE000EFF4`.

Secure software can access the Non-secure version of this register via DCIDR1_NS located at `0xE002EFF4`. The location `0xE002EFF4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

## D1.2.28 DCIDR2, SCS Component Identification Register 2

The DCIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

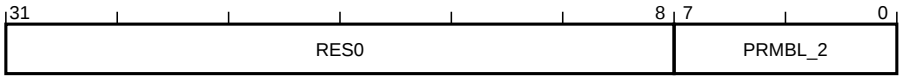
**Attributes**
32-bit read-only register located at `0xE000EFF8`.

Secure software can access the Non-secure version of this register via DCIDR2_NS located at `0xE002EFF8`. The location `0xE002EFF8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

## D1.2.29 DCIDR3, SCS Component Identification Register 3

The DCIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

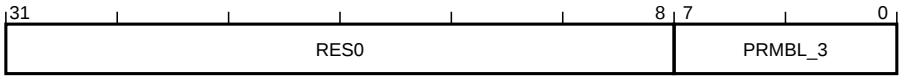
**Attributes**
32-bit read-only register located at `0xE000EFFC`.

Secure software can access the Non-secure version of this register via DCIDR3_NS located at `0xE002EFFC`. The location `0xE002EFFC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DCIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

### D1.2.30 DCIMVAC, Data Cache line Invalidate by Address to PoC

The DCIMVAC characteristics are:

**Purpose**
Invalidate data or unified cache line by address to PoC.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

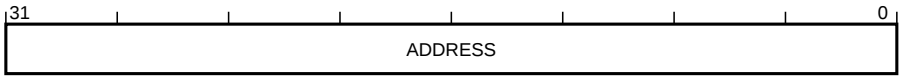
**Attributes**
32-bit write-only register located at `0xE000EF5C`.

Secure software can access the Non-secure version of this register via DCIMVAC_NS located at `0xE002EF5C`. The location `0xE002EF5C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCIMVAC bit assignments are:




**ADDRESS, bits [31:0]**
Address. Writing to this field initiates the maintenance operation for the address written.

### D1.2.31 DCISW, Data Cache line Invalidate by Set/Way

The DCISW characteristics are:

**Purpose**
Invalidate data or unified cache line by set/way.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

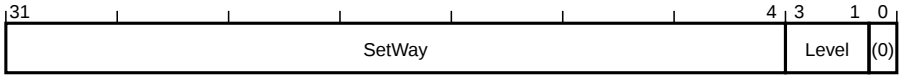
**Attributes**
32-bit write-only register located at `0xE000EF60`.

Secure software can access the Non-secure version of this register via DCISW_NS located at `0xE002EF60`. The location `0xE002EF60` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCISW bit assignments are:




**SetWay, bits [31:4]**
Cache set/way. Contains two fields: Way, bits[31:32-A], the number of the way to operate on. Set, bits[B-1:L], the number of the set to operate on. Bits[L-1:4] are RES0. A = $Log_2$(ASSOCIATIVITY), L = $Log_2$(LINELEN), B = (L + S), S = $Log_2$(NSETS). ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.

**Level, bits [3:1]**
Cache level. Cache level to operate on, minus 1. For example, this field is 0 for operations on L1 cache, or 1 for operations on L2 cache.

**Bit [0]**
Reserved, RES0.

### D1.2.32 DCRDR, Debug Core Register Data Register

The DCRDR characteristics are:

**Purpose**

With the DCRSR, provides debug access to the general-purpose registers, special-purpose registers, and the Floating-point Extension registers. If the Main Extension is implemented, it can also be used for message passing between an external debugger and a debug agent running on the PE.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then this register is accessible only to the debugger and UNKNOWN to software.

**Configurations**

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

**Attributes**

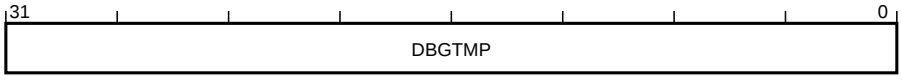
32-bit read/write register located at `0xE000EDF8`.

Secure software can access the Non-secure version of this register via DCRDR_NS located at `0xE002EDF8`. The location `0xE002EDF8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DCRDR bit assignments are:




**DBGTMP, bits [31:0]**

Data temporary buffer. Provides debug access for reading and writing the general-purpose registers, special-purpose registers, and Floating-point Extension registers.

The value of this register is UNKNOWN if the PE is in Debug state, the debugger has written to DCRSR since entering Debug state and DHCSR.S_REGRDY is set to 0. The value of this register is UNKNOWN if the Main Extension is not implemented and the PE is in Non-debug state.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.33 DCRSR, Debug Core Register Select Register

The DCRSR characteristics are:

**Purpose**

With the DCRDR, provides debug access to the general-purpose registers, special-purpose registers, and the Floating-point Extension registers. A write to the DCRSR specifies the register to transfer, whether the transfer is a read or write, and starts the transfer.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Writes to this register while the PE is in Non-debug state are ignored.

This register is accessible only to the debugger and RES0 to software.

**Configurations**

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

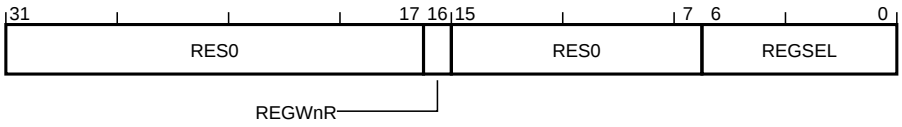
**Attributes**

32-bit write-only register located at `0xE000EDF4`.

This register is not banked between Security states.

## Field descriptions

The DCRSR bit assignments are:




**Bits [31:17]**

Reserved, RES0.

**REGWnR, bit [16]**

Register write/not-read. Specifies the access type for the transfer.

The possible values of this bit are:

**0**

Read.

**1**

Write.

**Bits [15:7]**

Reserved, RES0.

**REGSEL, bits [6:0]**

Register selector. Specifies the general-purpose register, special-purpose register, or Floating-point Extension register to transfer.

The possible values of this field are:

**0b0000000-0b0001100**
General-purpose registers R0-R12.

**0b0001101**
Current stack pointer, SP.

**0b0001110**
LR.

**0b0001111**
DebugReturnAddress.

**0b0010000**
XPSR / EAPSR, if privileged debug is permitted for the current state this values accesses XPSR, otherwise EAPSR is accessed.

**0b0010001**
Current state main stack pointer, SP_main.

Accessible only when privileged debug is permitted for the current state.

**0b0010010**
Current state process stack pointer, SP_process.

**0b0010100**
Current state {CONTROL[7:0],FAULTMASK[7:0],BASEPRI[7:0],PRIMASK[7:0]}.

If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0.

Accessible only when privileged debug is permitted for the current state.

**0b0011000**
Non-secure main stack pointer, MSP_NS.

If the Security Extension is not implemented, this value is reserved.

Accessible only when privileged debug is permitted for the Non-secure state.

**0b0011001**
Non-secure process stack pointer, PSP_NS.

If the Security Extension is not implemented, this value is reserved.

**0b0011010**
Secure main stack pointer, MSP_S. Accessible only when DHCSR.S_SDE == 1.

If the Security Extension is not implemented, this value is reserved.

Accessible only when privileged debug is permitted for the Secure state.

**0b0011011**
Secure process stack pointer, PSP_S. Accessible only when DHCSR.S_SDE == 1.

If the Security Extension is not implemented, this value is reserved.

**0b0011100**
Secure main stack limit, MSPLIM_S. Accessible only when DHCSR.S_SDE == 1.

If the Security Extension is not implemented, this value is reserved.

Accessible only when privileged debug is permitted for the Secure state.

**0b0011101**
Secure process stack limit, PSPLIM_S. Accessible only when DHCSR.S_SDE == 1.

If the Security Extension is not implemented, this value is reserved.

**0b0011110**
Non-secure main stack limit, MSPLIM_NS.

If the Main Extension is not implemented, this value is reserved.

Accessible only when privileged debug is permitted for the Non-secure state.

**0b0011111**
Non-secure process stack limit, PSPLIM_NS.

If the Main Extension is not implemented, this value is reserved.

**0b0100001**
FPSCR.

If MVE and the Floating-point Extension are not implemented, this value is reserved.

**0b0100010**
{CONTROL_S[7:0],FAULTMASK_S[7:0],BASEPRI_S[7:0],PRIMASK_S[7:0]}. Accessible only when DHCSR.S_SDE == 1 and privileged debug is permitted for the Secure state.

If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0. If the Security Extension is not implemented, this value is reserved.

**0b0100011**
{CONTROL_NS[7:0],FAULTMASK_NS[7:0],BASEPRI_NS[7:0],PRIMASK_NS[7:0]}. Accessible only when privileged debug is permitted for the Non-secure state.

If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0. If the Security Extension is not implemented, this value is reserved.

**0b0100100**
VPR.

If MVE is not implemented, this value is reserved.

**0b1000000-0b1011111**
FP registers, S0-S31.

If MVE and the Floating-point Extension are not implemented, these values are reserved.

All other values are reserved.

If the Security Extension and either MVE or the Floating-point Extension are implemented, then FPSCR, VPR and S0-S31 are not accessible:

- From Non-secure state if DHCSR.S_SDE == 0 and FPCCR indicates the registers contain values from Secure state.
- From Non-secure state if DHCSR.S_SDE == 0 and NSACR prevents Non-secure access to the registers.
- FPCCR.LSPACT, the banked variant associated with the Floating-point context as indicated by FPCCR_S.S, is set and only unprivileged debug is permitted for the security state associated with the Floating-point state (as indicated by FPCCR.S).

Registers that are not accessible are RAZ/WI.

If this field is written with a reserved value, the PE might behave as if a defined value was written, or ignore the value written, and the value of DCRDR becomes UNKNOWN.

### D1.2.34 DDEVARCH, SCS Device Architecture Register

The DDEVARCH characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

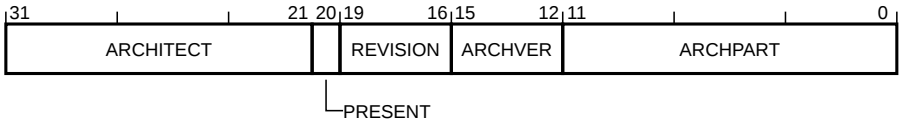
**Attributes**
32-bit read-only register located at `0xE000EFBC`.

Secure software can access the Non-secure version of this register via DDEVARCH_NS located at `0xE002EFBC`. The location `0xE002EFBC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DDEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**
Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

**`0x23B`**
JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

**PRESENT, bit [20]**
DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

**1**
DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**
Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**
M-profile debug architecture v3.0.

This field reads as `0b0000`.

**ARCHVER, bits [15:12]**
Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0010**
M-profile debug architecture v3.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0010`.

**ARCHPART, bits [11:0]**
Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA04**
M-profile debug architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA04`.

### D1.2.35 DDEVTYPE, SCS Device Type Register

The DDEVTYPE characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

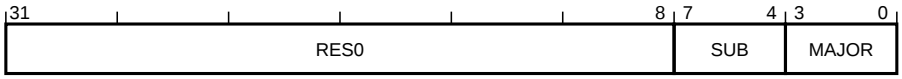
**Attributes**
32-bit read-only register located at `0xE000EFCC`.

Secure software can access the Non-secure version of this register via DDEVTYPE_NS located at `0xE002EFCC`. The location `0xE002EFCC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DDEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Sub-type. Component sub-type.

The possible values of this field are:

**`0x0`**
Other.

This field reads as `0b0000`.

**MAJOR, bits [3:0]**
Major type. CoreSight major type.

The possible values of this field are:

**0x0**
Miscellaneous.

This field reads as `0b0000`.

## D1.2.36 DEMCR, Debug Exception and Monitor Control Register

The DEMCR characteristics are:

**Purpose**
Manages vector catch behavior and DebugMonitor handling when debugging.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If Armv8.1-M is implemented, this register is word, halfword and byte accessible.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

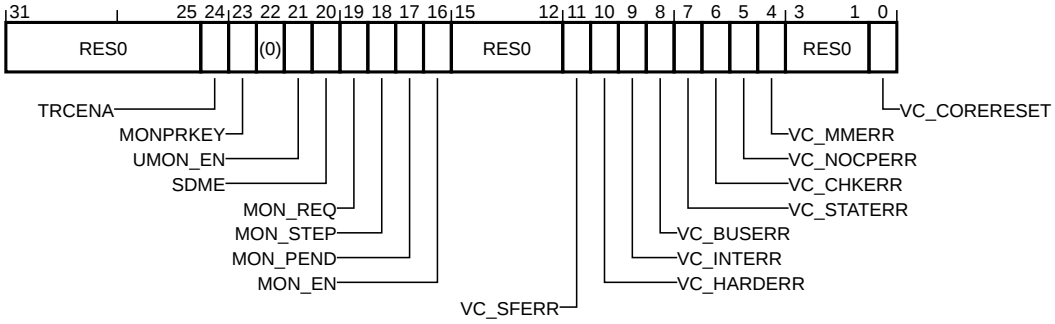
**Attributes**
32-bit read/write register located at `0xE000EDFC`.

Secure software can access the Non-secure version of this register via DEMCR_NS located at `0xE002EDFC`. The location `0xE002EDFC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DEMCR bit assignments are:




**Bits [31:25]**
Reserved, RES0.

**TRCENA, bit [24]**
Trace enable. Global enable for all DWT, PMU, and ITM features.

The possible values of this bit are:

**0**
DWT, PMU, and ITM features disabled.

**1**
DWT, PMU, and ITM features enabled.

If the DWT, PMU, and ITM units are not implemented, this bit is RES0. See the descriptions of DWT, PMU, and ITM for details of which features this bit controls.

Setting this bit to 0 might not stop all events. To ensure that all events are stopped, software must set all DWT, PMU, and ITM feature enable bits to 0, and ensure that all trace generated by the DWT, PMU, and ITM has been flushed, before setting this bit to 0.

It is IMPLEMENTATION DEFINED whether this bit affects how the system processes trace.

Arm recommends that this bit is set to 1 when using an ETM even if any implemented DWT, PMU, and ITM are not being used.

This bit resets to zero on a Cold reset.

**MONPRKEY, bit [23]**
Monitor pend req key. Writes to the MON_PEND and MON_REQ fields are ignored unless MONPRKEY is concurrently written to zero.

The possible values of this bit are:

**0**
Concurrent write to MON_PEND and MON_REQ are not ignored.

**1**
Concurrent write to MON_PEND and MON_REQ are ignored.

This bit reads-as-zero.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

**Bit [22]**
Reserved, RES0.

**UMON_EN, bit [21]**
Unprivileged monitor enable. DebugMonitor pend enable when the PE is in an unprivileged mode.

The possible values of this bit are:

**0b0**
DebugMonitor exception controlled by DEMCR.MON_EN.

**0b1**
DebugMonitor exception can be pended for unprivileged execution.

Writes to this field by unprivileged DAP accesses are ignored.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If UDE is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**SDME, bit [20]**
Secure DebugMonitor enable. Indicates whether the DebugMonitor targets the Secure or the Non-secure state and whether debug events are allowed in Secure state.

The possible values of this bit are:

**0**
Debug events prohibited in Secure state and the DebugMonitor exception targets Non-secure state.

**1**
Debug events allowed in Secure state and the DebugMonitor exception targets Secure state.

When DebugMonitor exception is not pending or active, this bit reflects the value of SecureDebugMonitorAllowed(), otherwise, the previous value is retained.

This bit is read-only.

If the Security Extension is not implemented, this bit is RES0.

If the Main Extension is not implemented, this bit is RES0.

**MON_REQ, bit [19]**
Monitor request. DebugMonitor semaphore bit.

The PE does not use this bit. The monitor software defines the meaning and use of this bit.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MON_STEP, bit [18]**
Monitor step. Enable DebugMonitor exception stepping.

The possible values of this bit are:

**0**
Stepping disabled.

**1**
Stepping enabled.

The effect of changing this bit at an execution priority that is lower than the priority of the DebugMonitor exception is UNPREDICTABLE. Unprivileged writes to this bit from the DAP are ignored.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MON_PEND, bit [17]**
Monitor pend. Sets or clears the pending state of the DebugMonitor exception.

The possible values of this bit are:

**0**
Clear the status of the DebugMonitor exception to not pending.

**1**
Set the status of the DebugMonitor exception to pending.

When the DebugMonitor exception is pending it becomes active subject to the exception priority rules. The effect of setting this bit to 1 is not affected by the value of the MON_EN and UMON_EN bits. This means that software or a debugger can set MON_PEND to 1 and pend a DebugMonitor exception, even when MON_EN and UMON_EN are set to 0.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MON_EN, bit [16]**
Monitor enable. Enable the DebugMonitor exception.

The possible values of this bit are:

**0**
DebugMonitor exception disabled.

**1**
DebugMonitor exception enabled.

If a debug event halts the PE, the PE ignores the value of this bit. Unprivileged writes to this bit from the DAP are ignored.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Bits [15:12]**
Reserved, RES0.

**VC_SFERR, bit [11]**
Vector Catch SecureFault. SecureFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**
Halting debug trap on SecureFault disabled.

**1**
Halting debug trap on SecureFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, HaltingDebugAllowed() == FALSE, DHCSR.S_SDE == 0, or DHCSR.S_SUIDE == 1.

If the Security Extension is not implemented, this bit is RES0.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_HARDERR, bit [10]**
Vector Catch HardFault errors. HardFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**
Halting debug trap on HardFault disabled.

**1**
Halting debug trap on HardFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_INTERR, bit [9]**
Vector Catch interrupt errors. Enable Halting debug vector catch for faults arising from lazy state preservation, stack violations and context stacking or unstacking during exception entry or return.

The possible values of this bit are:

**0**
Halting debug trap on faults disabled.

**1**
Halting debug trap on faults enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_BUSERR, bit [8]**
Vector Catch BusFault errors. BusFault exception Halting debug vector catch enable.

The possible values of this bit are:

**0**
Halting debug trap on BusFault disabled.

**1**
Halting debug trap on BusFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_STATERR, bit [7]**
Vector Catch state errors. Enable Halting debug trap on a UsageFault exception caused by a state information error, for example an Undefined Instruction exception.

The possible values of this bit are:

**0**
Halting debug trap on UsageFault caused by state information error disabled.

**1**
Halting debug trap on UsageFault caused by state information error enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_CHKERR, bit [6]**
Vector Catch check errors. Enable Halting debug trap on a UsageFault exception caused by an alignment check error or divide-by-zero trap.

The possible values of this bit are:

**0**
Halting debug trap on UsageFault caused by checking error disabled.

**1**
Halting debug trap on UsageFault caused by checking error enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_NOCPERR, bit [5]**
Vector Catch NOCP errors. Enable Halting debug trap on a UsageFault caused by an access to a coprocessor.

The possible values of this bit are:

**0**
Halting debug trap on UsageFault caused by access to a coprocessor disabled.

**1**
Halting debug trap on UsageFault caused by access to a coprocessor enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**VC_MMERR, bit [4]**
Vector Catch MemManage errors. Enable Halting debug trap on a MemManage exception.

The possible values of this bit are:

**0**
Halting debug trap on MemManage disabled.

**1**
Halting debug trap on MemManage enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, DHCSR.S_NSUIDE == 1, or HaltingDebugAllowed() == FALSE. If the Security Extension is implemented, the PE also ignores the value of this bit if DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**Bits [3:1]**
Reserved, RES0.

**VC_CORERESET, bit [0]**
Vector Catch Core reset. Enable Reset Vector Catch. This causes a Warm reset to halt a running system.

The possible values of this bit are:

**0**
Halting debug trap on reset disabled.

**1**
Halting debug trap on reset enabled.

If DHCSR.C_DEBUGEN == 0, the PE ignores the value of this bit. Otherwise, when this bit is set to 1 a Warm reset will pend a Vector Catch debug event. The debug event is pended regardless of debug permissions or the security state of the PE, and the PE will halt when it enters a mode or state where debug is enabled.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

## D1.2.37 DFSR, Debug Fault Status Register

The DFSR characteristics are:

**Purpose**
Shows which debug event occurred.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

**Attributes**
32-bit read/write-one-to-clear register located at `0xE000ED30`.

Secure software can access the Non-secure version of this register via DFSR_NS located at `0xE002ED30`. The location `0xE002ED30` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DFSR bit assignments are:


**Bits [31:6]**
Reserved, RES0.

**PMU, bit [5]**
PMU event. Sticky flag indicating whether a PMU counter overflow event has occurred.

The possible values of this bit are:

**0**
PMU event has not occurred.

**1**
PMU event has occurred.

This bit resets to zero on a Cold reset.

**EXTERNAL, bit [4]**
External event. Sticky flag indicating whether an External debug request debug event has occurred.

The possible values of this bit are:

**0**
Debug event has not occurred.

**1**
Debug event has occurred.

This bit resets to zero on a Cold reset.

**VCATCH, bit [3]**
Vector Catch event. Sticky flag indicating whether a Vector catch debug event has occurred.

The possible values of this bit are:

**0**
Debug event has not occurred.

**1**
Debug event has occurred.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**DWTTRAP, bit [2]**
Watchpoint event. Sticky flag indicating whether a Watchpoint debug event has occurred.

The possible values of this bit are:

**0**
Debug event has not occurred.

**1**
Debug event has occurred.

If the DWT is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**BKPT, bit [1]**
Breakpoint event. Sticky flag indicating whether a Breakpoint debug event has occurred.

The possible values of this bit are:

**0**
Debug event has not occurred.

**1**
Debug event has occurred.

This bit resets to zero on a Cold reset.

**HALTED, bit [0]**
Halt or step event. Sticky flag indicating that a Halt request debug event or Step debug event has occurred.

The possible values of this bit are:

**0**
Debug event has not occurred.

**1**
Debug event has occurred.

This bit resets to zero on a Cold reset.

## D1.2.38 DHCSR, Debug Halting Control and Status Register

The DHCSR characteristics are:

**Purpose**
Controls Halting debug.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

It is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software.

**Configurations**
Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

**Attributes**
32-bit read/write register located at `0xE000EDF0`.

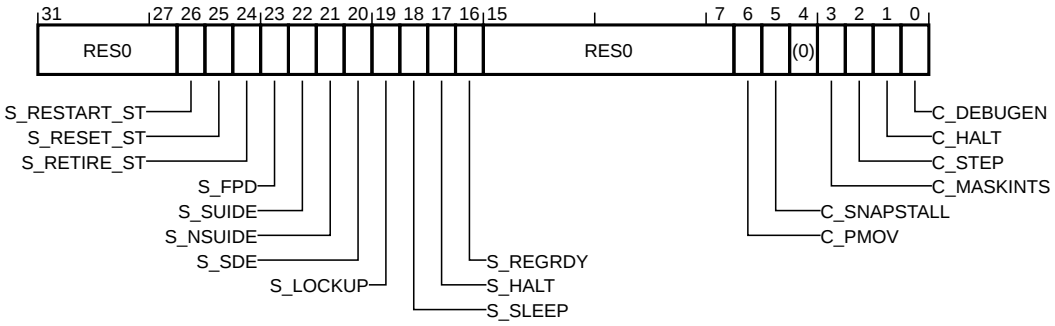
Secure software can access the Non-secure version of this register via DHCSR_NS located at `0xE002EDF0`. The location `0xE002EDF0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

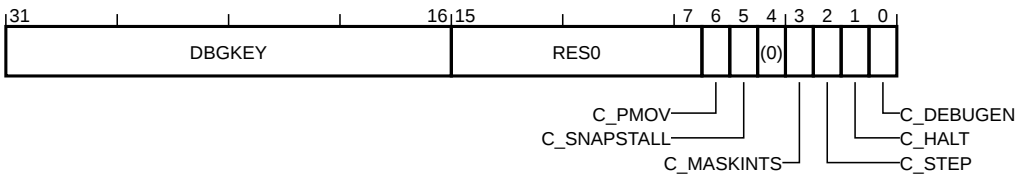
### Field descriptions

The DHCSR bit assignments are:

On a read:




On a write:




**DBGKEY, bits [31:16], on a write**
Debug key. A debugger must write `0xA05F` to this field to enable write access to the remaining bits, otherwise the PE ignores the write access.

The possible values of this field are:

**0xA05F**
Writes accompanied by this value update bits[15:0].

**Not 0xA05F**
Write ignored.

**Bits [31:27], on a read**
Reserved, RES0.

**S_RESTART_ST, bit [26], on a read**
Restart sticky status. Indicates the PE has processed a request to clear DHCSR.C_HALT to 0. That is, either a write to DHCSR that clears DHCSR.C_HALT from 1 to 0, or an External Restart Request.

The possible values of this bit are:

**0**
PE has not left Debug state since the last read of DHCSR.

**1**
PE has left Debug state since the last read of DHCSR.

If the PE is not halted when C_HALT is cleared to zero, it is UNPREDICTABLE whether this bit is set to 1. If DHCSR.C_DEBUGEN == 0 this bit reads as an UNKNOWN value.

This bit clears to zero when read.

**Note**

If the request to clear C_HALT is made simultaneously with a request to set C_HALT, for example a restart request and external debug request occur together, then the PE notionally leaves Debug state and immediately halts again and S_RESTART_ST is set to 1.

**S_RESET_ST, bit [25], on a read**
Reset sticky status. Indicates whether the PE has been reset since the last read of the DHCSR.

The possible values of this bit are:

**0**
No reset since last DHCSR read.

**1**
At least one reset since last DHCSR read.

This bit clears to zero when read.

This bit resets to one on a Warm reset.

**S_RETIRE_ST, bit [24], on a read**
Retire sticky status. Set to 1 every time the PE retires one of more instructions.

The possible values of this bit are:

**0**
No instruction retired since last DHCSR read.

**1**
At least one instruction retired since last DHCSR read.

This bit clears to zero when read.

This bit resets to an UNKNOWN value on a Warm reset.

**S_FPD, bit [23], on a read**
Floating-point registers Debuggable. Indicates that FPSCR, VPR, and the Floating-point registers are RAZ/WI in the current PE state when accessed via DCRSR. This reflects CanDebugAccessFP().

The possible values of this bit are:

**0b0**
Floating-point registers accessible.

**0b1**
Floating-point registers are RAZ/WI.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

**S_SUIDE, bit [22], on a read**
Secure unprivileged halting debug enabled. Indicates whether Secure unprivileged-only halting debug is allowed or active.

The possible values of this bit are:

**0**
Secure invasive halting debug prohibited or not restricted to an unprivileged mode.

**1**
Unprivileged Secure invasive halting debug enabled.

If the PE is in Non-debug state, this bit reflects the value of UnprivHaltingDebugAllowed(TRUE) && !SecureHaltingDebugAllowed().

The value of this bit does not change whilst the PE remains in Debug state.

If the Security Extension is not implemented, this bit is RES0.

If UDE is not implemented, this bit is RES0.

**S_NSUIDE, bit [21], on a read**
Non-secure unprivileged halting debug enabled. Indicates whether Non-secure unprivileged-only halting debug is allowed or active.

The possible values of this bit are:

**0**
Non-secure invasive halting debug prohibited or not restricted to an unprivileged mode.

**1**
Unprivileged Non-secure invasive halting debug enabled.

If the PE is in Non-debug state, this bit reflects the value of UnprivHaltingDebugAllowed(FALSE) && !HaltingDebugAllowed().

The value of this bit does not change whilst the PE remains in Debug state.

If UDE is not implemented, this bit is RES0.

**S_SDE, bit [20], on a read**
Secure debug enabled. Indicates whether Secure invasive debug is allowed.

The possible values of this bit are:

**0**
Secure invasive debug prohibited.

**1**
Secure invasive debug allowed.

If the PE is in Non-debug state, this bit reflects the value of SecureHaltingDebugAllowed().

If the PE is in Debug state then this bit is 1 if the PE entered Debug state from either Non-secure state with SecureHaltingDebugAllowed() == TRUE or from Secure state, and 0 otherwise. The value of this bit does not change while the PE remains in Debug state.

If the Security Extension is not implemented, this bit is RES0.

**S_LOCKUP, bit [19], on a read**
Lockup status. Indicates whether the PE is in Lockup state.

The possible values of this bit are:

**0**
Not locked up.

**1**
Locked up.

This bit can only be read as 1 by a remote debugger, using the DAP. The value of 1 indicates that the PE is running but locked up. The bit clears to 0 when the PE enters Debug state.

**S_SLEEP, bit [18], on a read**
Sleeping status. Indicates whether the PE is sleeping.

The possible values of this bit are:

**0**
Not sleeping.

**1**
Sleeping.

The debugger must set the C_HALT bit to 1 to gain control, or wait for an interrupt or other wakeup event to wakeup the system.

**S_HALT, bit [17], on a read**
Halted status. Indicates whether the PE is in Debug state.

The possible values of this bit are:

**0**
In Non-debug state.

**1**
In Debug state.

**S_REGRDY, bit [16], on a read**
Register ready status. Handshake flag to transfers through the DCRDR.

The possible values of this bit are:

**0**
Write to DCRSR performed, but transfer not yet complete.

**1**
Transfer complete, or no outstanding transfer.

This bit is valid only when the PE is in Debug state, otherwise this bit is UNKNOWN.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [15:7]**
Reserved, RES0.

**C_PMOV, bit [6]**
Halt on PMU overflow control. Request entry to Debug state when a PMU counter overflows.

The possible values of this bit are:

**0**
No action.

**1**

If C_DEBUGEN is set to 1, then when a PMU counter is configured to generate an interrupt overflows, the PE sets DHCSR.C_HALT to 1 and DFSR.PMU to 1.

PMU_OVSSET and PMU_OVSCLR indicate which counter or counters triggered the halt.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Cold reset.

**C_SNAPSTALL, bit [5]**

Snap stall control. Allow imprecise entry to Debug state.

The possible values of this bit are:

**0**

No action.

**1**

Allows imprecise entry to Debug state, for example by forcing any stalled load or store instruction to be abandoned.

Setting this bit to 1 allows a debugger to request an imprecise entry to Debug state. Writing 1 to this bit makes the state of the memory system UNPREDICTABLE. Therefore if a debugger writes 1 to this bit it must reset the system before leaving Debug state.

The effect of setting this bit to 1 is UNPREDICTABLE unless the DHCSR write also sets C_DEBUGEN and C_HALT to 1. This means that if the PE is not already in Debug state, it enters Debug state when the stalled instruction completes.

If the Security Extension is implemented, then writes to this bit are ignored when DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1.

If DHCSR.C_DEBUGEN == 0, HaltingDebugAllowed() == FALSE, or DHCSR.S_NSUIDE == 1, the PE ignores this bit and behaves as if it is set to 0.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**Note**

A debugger can write to the DHCSR to clear this bit to 0. However, this does not remove the UNPREDICTABLE state of the memory system caused by setting C_SNAPSTALL to 1. The architecture does not guarantee that setting this bit to 1 will force an entry to Debug State. Arm strongly recommends that a value of 1 is never written to C_SNAPSTALL when the PE is in Debug state. It is IMPLEMENTATION DEFINED whether this bit behaves as RAZ/WI.

**Bit [4]**

Reserved, RES0.

**C_MASKINTS, bit [3]**

Mask interrupts control. When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts.

The possible values of this bit are:

**0**

Do not mask.

**1**

Mask PendSV, SysTick and external configurable interrupts.

The effect of any single write to DHCSR that changes the value of this bit is UNPREDICTABLE unless one of:

- Before the write, DHCSR.{S_HALT,C_HALT} are both set to 1 and the write also writes 1 to DHCSR.C_HALT.
- Before the write, DHCSR.C_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE, and the write writes 0 to DHCSR.C_MASKINTS.

This means that a single write to DHCSR must not clear DHCSR.C_HALT to 0 and change the value of the C_MASKINTS bit.

If the Security Extension is implemented and either DHCSR.S_SDE == 0 or DHCSR.S_SUIDE == 1, this bit does not affect interrupts targeting Secure state.

If DHCSR.C_DEBUGEN == 0, HaltingDebugAllowed() == FALSE, or DHCSR.S_NSUIDE == 1, the PE ignores this bit and behaves as if it is set to 0.

If DHCSR.C_DEBUGEN == 0 this bit reads as an UNKNOWN value.

This bit resets to an UNKNOWN value on a Cold reset.

**Note**

This bit does not affect NMI.

**C_STEP, bit [2]**

Step control. Enable single instruction step.

The possible values of this bit are:

**0**

No effect.

**1**

Single step enabled.

The effect of a single write to DHCSR that changes the value of this bit is UNPREDICTABLE unless one of:

- Before the write, DHCSR.{S_HALT,C_HALT} are both set to 1.
- Before the write, DHCSR.C_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE, and the write writes 0 to DHCSR.C_STEP.

The PE ignores this bit and behaves as if it set to 0 if any of:

- DHCSR.C_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE.
- The Security Extension is implemented, DHCSR.S_SDE == 0 and the PE is in Secure state.

If DHCSR.C_DEBUGEN == 0 this bit reads as an UNKNOWN value.

This bit resets to an UNKNOWN value on a Cold reset.

**C_HALT, bit [1]**

Halt control. PE to enter Debug state halt request.

The possible values of this bit are:

**0**

Causes the PE to leave Debug state, if the PE is in Debug state.

**1**

Halt the PE.

The PE sets C_HALT to 1 when a debug event pends an entry to Debug state.

The PE ignores this bit and behaves as if it is set to 0 if any of:

- UnprivHaltingDebugAllowed(IsSecure()) == FALSE and either DHCSR.C_DEBUGEN == 0 or HaltingDebugAllowed() == FALSE.

- The Security Extension is implemented, DHCSR.S_SDE == 0 and the PE is in Secure state.

If DHCSR.C_DEBUGEN == 0 this bit reads as an UNKNOWN value.

This bit resets to zero on a Warm reset.

**C_DEBUGEN, bit [0]**
Debug enable control. Enable Halting debug.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

If a debugger writes to DHCSR to change the value of this bit from 0 to 1, it must also write 0 to the C_MASKINTS bit, otherwise behavior is UNPREDICTABLE.

If this bit is set to 0:

- The PE behaves as if DHCSR.{C_MASKINTS, C_STEP, C_HALT} are all set to 0.
- DHCSR.{S_RESTART_ST, C_MASKINTS, C_STEP, C_HALT} are UNKNOWN on reads of DHCSR.

This bit is read/write to the debugger. Writes from software are ignored.

This bit resets to zero on a Cold reset.

### D1.2.39 DLAR, SCS Software Lock Access Register

The DLAR characteristics are:

**Purpose**

Provides CoreSight Software Lock control for the SCS, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

**Attributes**

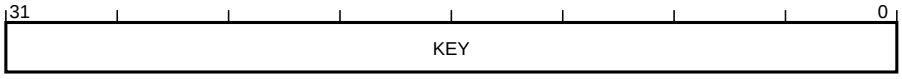
32-bit write-only register located at `0xE000EFB0`.

Secure software can access the Non-secure version of this register via DLAR_NS located at `0xE002EFB0`. The location `0xE002EFB0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DLAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

## D1.2.40 DLSR, SCS Software Lock Status Register

The DLSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the SCS, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

**Attributes**

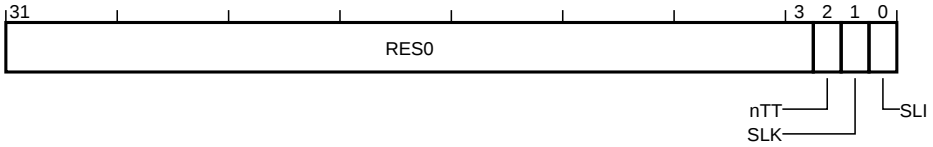
32-bit read-only register located at `0xE000EFB4`.

Secure software can access the Non-secure version of this register via DLSR_NS located at `0xE002EFB4`. The location `0xE002EFB4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DLSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of the component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Warm reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

### D1.2.41 DPIDR0, SCS Peripheral Identification Register 0

The DPIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

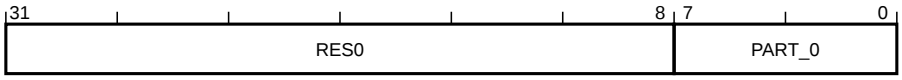
**Attributes**
32-bit read-only register located at `0xE000EFE0`.

Secure software can access the Non-secure version of this register via DPIDR0_NS located at `0xE002EFE0`. The location `0xE002EFE0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.42 DPIDR1, SCS Peripheral Identification Register 1

The DPIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

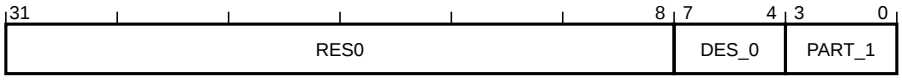
**Attributes**
32-bit read-only register located at `0xE000EFE4`.

Secure software can access the Non-secure version of this register via DPIDR1_NS located at `0xE002EFE4`. The location `0xE002EFE4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.43 DPIDR2, SCS Peripheral Identification Register 2

The DPIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

**Attributes**
32-bit read-only register located at `0xE000EFE8`.

Secure software can access the Non-secure version of this register via DPIDR2_NS located at `0xE002EFE8`. The location `0xE002EFE8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DPIDR2 bit assignments are:


**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

**DES_1, bits [2:0]**
JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.44 DPIDR3, SCS Peripheral Identification Register 3

The DPIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

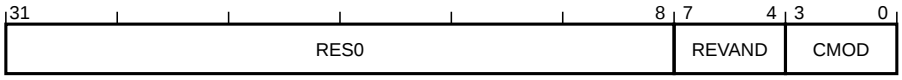
**Attributes**
32-bit read-only register located at `0xE000EFEC`.

Secure software can access the Non-secure version of this register via DPIDR3_NS located at `0xE002EFEC`. The location `0xE002EFEC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.45 DPIDR4, SCS Peripheral Identification Register 4

The DPIDR4 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

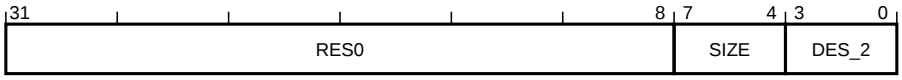
**Attributes**
32-bit read-only register located at `0xE000EFD0`.

Secure software can access the Non-secure version of this register via DPIDR4_NS located at `0xE002EFD0`. The location `0xE002EFD0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DPIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

**DES_2, bits [3:0]**
JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.46 DPIDR5, SCS Peripheral Identification Register 5

The DPIDR5 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

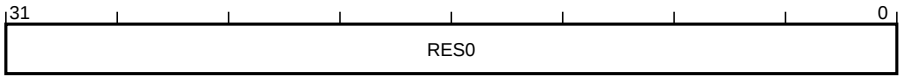
**Attributes**
32-bit read-only register located at `0xE000EFD4`.

Secure software can access the Non-secure version of this register via DPIDR5_NS located at `0xE002EFD4`. The location `0xE002EFD4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.47 DPIDR6, SCS Peripheral Identification Register 6

The DPIDR6 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

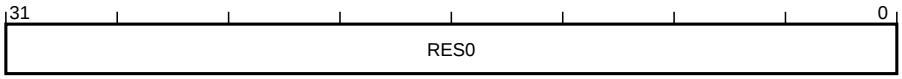
**Attributes**
32-bit read-only register located at `0xE000EFD8`.

Secure software can access the Non-secure version of this register via DPIDR6_NS located at `0xE002EFD8`. The location `0xE002EFD8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR6 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.48 DPIDR7, SCS Peripheral Identification Register 7

The DPIDR7 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the SCS.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

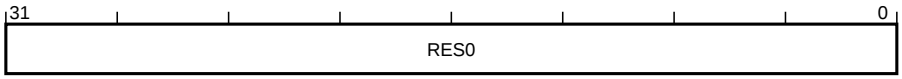
**Attributes**
32-bit read-only register located at `0xE000EFDC`.

Secure software can access the Non-secure version of this register via DPIDR7_NS located at `0xE002EFDC`. The location `0xE002EFDC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The DPIDR7 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

## D1.2.49 DSCEMCR, Debug Set Clear Exception and Monitor Control Register

The DSCEMCR characteristics are:

**Purpose**
Atomically sets or clears selected fields in the DEMCR register.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

Present only if Armv8.1-M is implemented.

This register is RES0 if Armv8.1-M is not implemented.

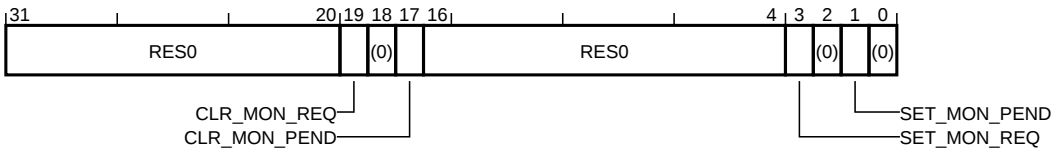
**Attributes**
32-bit write-only register located at `0xE000EE00`.

Secure software can access the Non-secure version of this register via DSCEMCR_NS located at `0xE002EE00`. The location `0xE002EE00` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The DSCEMCR bit assignments are:




**Bits [31:20]**
Reserved, RES0.

**CLR_MON_REQ, bit [19]**
Clear monitor request. Atomically clears the DEMCR.MON_REQ field.

The possible values of this bit are:

**0**
No effect.

**1**
Clear DEMCR.MON_REQ.

A write to this register with both SET_MON_REQ and CLR_MON_REQ set to 1 causes DEMCR.MON_REQ to become UNKNOWN.

**Bit [18]**
Reserved, RES0.

**CLR_MON_PEND, bit [17]**
Clear monitor pend. Atomically clears the DEMCR.MON_PEND field.

The possible values of this bit are:

**0**
No effect.

**1**
Clear DEMCR.MON_PEND.

A write to this register with both SET_MON_PEND and CLR_MON_PEND set to 1 causes DEMCR.MON_PEND to become UNKNOWN.

**Bits [16:4]**
Reserved, RES0.

**SET_MON_REQ, bit [3]**
Set monitor request. Atomically sets the DEMCR.MON_REQ field.

The possible values of this bit are:

**0**
No effect.

**1**
Sets DEMCR.MON_REQ.

A write to this register with both SET_MON_REQ and CLR_MON_REQ set to 1 causes DEMCR.MON_REQ to become UNKNOWN.

**Bit [2]**
Reserved, RES0.

**SET_MON_PEND, bit [1]**
Set monitor pend. Atomically sets the DEMCR.MON_PEND field.

The possible values of this bit are:

**0**
No effect.

**1**
Sets DEMCR.MON_PEND.

A write to this register with both SET_MON_PEND and CLR_MON_PEND set to 1 causes DEMCR.MON_PEND to become UNKNOWN.

**Bit [0]**
Reserved, RES0.

### D1.2.50 DSCSR, Debug Security Control and Status Register

The DSCSR characteristics are:

**Purpose**
Provides control and status information for Secure debug.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

This register is accessible only to the debugger and RES0 to software.

**Configurations**
Present only if the Security Extension is implemented.

This register is RES0 if the Security Extension is not implemented.

Present only if Halting debug is implemented.

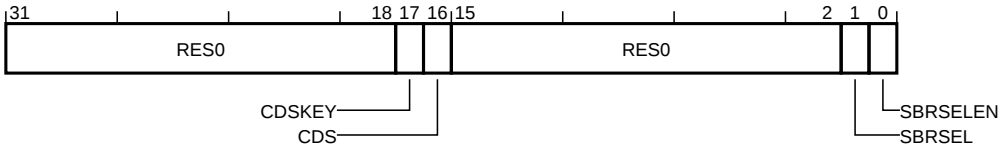
This register is RES0 if Halting debug is not implemented.

**Attributes**
32-bit read/write register located at `0xE000EE08`.

This register is not banked between Security states.

## Field descriptions

The DSCSR bit assignments are:




**Bits [31:18]**
Reserved, RES0.

**CDSKEY, bit [17]**
CDS write-enable key. Writes to the CDS bit are ignored unless CDSKEY is concurrently written to zero.

The possible values of this bit are:

**0**
Concurrent write to CDS not ignored.

**1**
Concurrent write to CDS ignored.

This bit reads-as-one.

**CDS, bit [16]**
Current domain Secure. This field indicates the current Security state of the processor.

The possible values of this bit are:

**0**
PE is in Non-secure state.

**1**
PE is in Secure state.

This bit is only writable if DHCSR.S_SDE is 1, either DHCSR.S_SUIDE == 0 or CONTROL.nPRIV == 1 for the state specified by the CDS value being written, the PE is halted in Debug state, and CDSKEY is concurrently written to zero.

**Bits [15:2]**
Reserved, RES0.

**SBRSEL, bit [1]**
Secure banked register select. If SBRSELEN is 1 this bit selects whether the Non-secure or the Secure versions of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

**0**
Selects the Non-secure versions.

**1**
Selects the Secure versions.

This bit behaves as RAZ/WI if DHCSR.S_SDE is 0.

This bit resets to zero on a Cold reset.

**SBRSELEN, bit [0]**
Secure banked register select enable. Controls whether the SBRSEL field or the current Security state of the processor selects which version of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

**0**
The current Security state of the PE determines which memory-mapped Banked registers are accessed by the debugger.

**1**
DSCSR.SBRSEL selects which memory-mapped Banked registers are accessed by the debugger.

This bit behaves as RAO/WI if DHCSR.S_SDE is 0.

This bit resets to zero on a Cold reset.

**Note**

This method of banked register selection means that the register aliasing is not used for accesses from the debugger. Accesses to the aliased addresses from the debugger have the same behavior as reserved addresses.

### D1.2.51 DWT_CIDR0, DWT Component Identification Register 0

The DWT_CIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

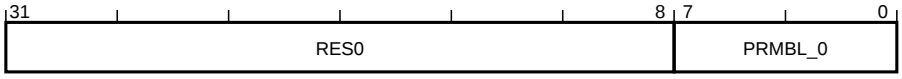
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FF0`.

This register is not banked between Security states.

## Field descriptions

The DWT_CIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_0, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

### D1.2.52 DWT_CIDR1, DWT Component Identification Register 1

The DWT_CIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

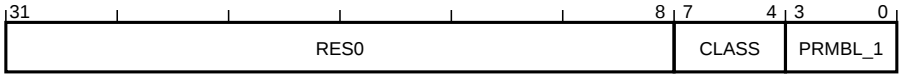
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FF4`.

This register is not banked between Security states.

## Field descriptions

The DWT_CIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

## D1.2.53 DWT_CIDR2, DWT Component Identification Register 2

The DWT_CIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

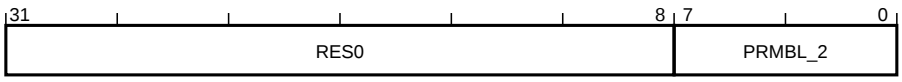
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FF8`.

This register is not banked between Security states.

### Field descriptions

The DWT_CIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

## D1.2.54 DWT_CIDR3, DWT Component Identification Register 3

The DWT_CIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

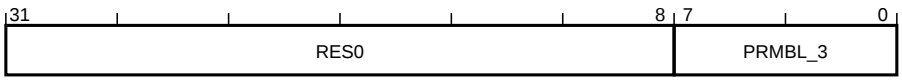
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FFC`.

This register is not banked between Security states.

### Field descriptions

The DWT_CIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

### D1.2.55 DWT_COMPn, DWT Comparator Register, n = 0 - 14

The DWT_COMP{0..14} characteristics are:

**Purpose**

Provides a reference value for use by watchpoint comparator *n*.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

**Attributes**

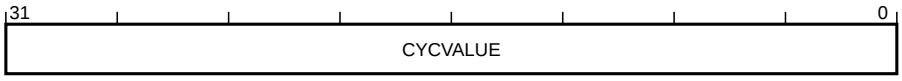
32-bit read/write register located at `0xE0001020` + 16*n*.

This register is not banked between Security states.

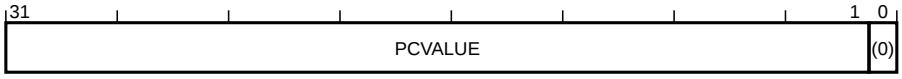
## Field descriptions

The DWT_COMP{0..14} bit assignments are:

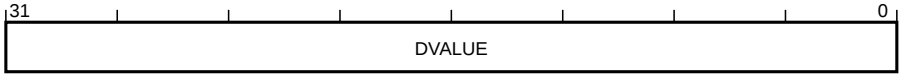
**When DWT_FUNCTIONn.MATCH == 0b0001**:



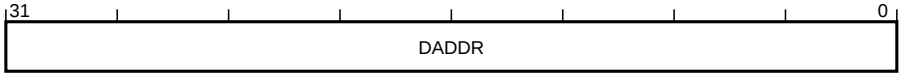

**When DWT_FUNCTIONn.MATCH == 0b001x**:




**When DWT_FUNCTIONn.MATCH == 0b10xx**:




**When DWT_FUNCTIONn.MATCH == 0bx1xx**:



**CYCVALUE, bits [31:0], when DWT_FUNCTIONn.MATCH == 0b0001**
Cycle value. Reference value for comparison with cycle count.

This field resets to an UNKNOWN value on a Cold reset.

**PCVALUE, bits [31:1], when DWT_FUNCTIONn.MATCH == 0b001x**
PC value. Reference value for comparison with Program Counter.

This field resets to an UNKNOWN value on a Cold reset.

**Bit [0], when DWT_FUNCTIONn.MATCH == 0b001x**
Reserved, RES0.

**DADDR, bits [31:0], when DWT_FUNCTIONn.MATCH == 0bx1xx**
Data address. Reference value for comparison with load or store address.

For halfword address comparisons, DADDR[0] is RES0. For byte address comparisons, DADDR[1:0] are RES0.

This field resets to an UNKNOWN value on a Cold reset.

**DVALUE, bits [31:0], when DWT_FUNCTIONn.MATCH == 0b10xx**
Data value. Reference value for comparison with load or store data.

For halfword or word comparisons, the data value is in little-endian order. That is, the least significant byte of this register is compared with the byte targeting the lowest address in memory.

For byte or halfword comparisons, if the value of the byte or halfword is not replicated across all byte or halfword lanes, the value used for the comparison is UNKNOWN.

This field resets to an UNKNOWN value on a Cold reset.

## D1.2.56 DWT_CPICNT, DWT CPI Count Register

The DWT_CPICNT characteristics are:

**Purpose**
Counts additional cycles required to execute multicycle instructions and instruction fetch stalls.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

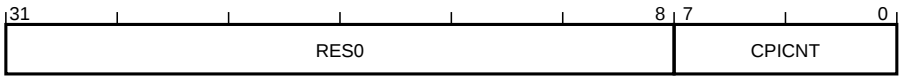
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

**Attributes**
32-bit read/write register located at `0xE0001008`.

This register is not banked between Security states.

### Field descriptions

The DWT_CPICNT bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CPICNT, bits [7:0]**
Base instruction overhead counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.CPIEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed.
- No load-store operation is in progress, see DWT_LSUCNT.
- No exception-entry or exception-exit operation is in progress, see DWT_EXCCNT.
- The PE is not in a power-saving mode, see DWT_SLEEPCNT.
- Either SecureNoninvasiveDebugAllowed() == TRUE, or the PE is in Non-secure state and NoninvasiveDebugAllowed() == TRUE.

The definition of "no instruction is executed" is IMPLEMENTATION DEFINED. Arm recommends that this counts each cycle on which no instruction is retired.

Initialized to zero when the counter is disabled and DWT_CTRL.CPIEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

## D1.2.57 DWT_CTRL, DWT Control Register

The DWT_CTRL characteristics are:

**Purpose**
Provides configuration and status information for the DWT unit, and used to control features of the unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the DWT is implemented.

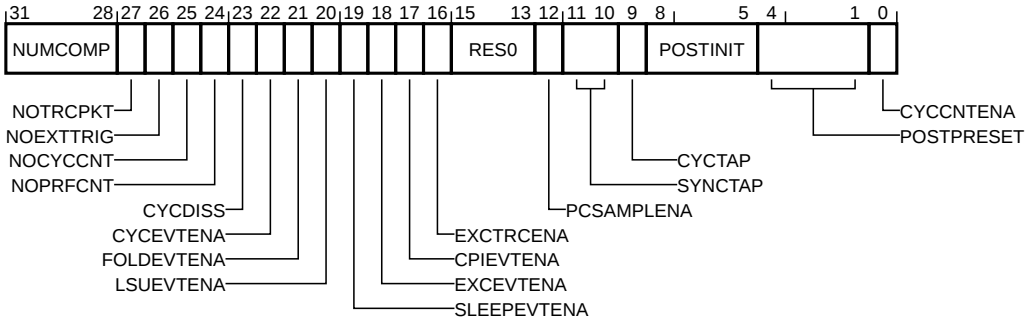
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read/write register located at `0xE0001000`.

This register is not banked between Security states.

### Field descriptions

The DWT_CTRL bit assignments are:




**NUMCOMP, bits [31:28]**
Number of comparators. Number of DWT comparators implemented.

A value of zero indicates no comparator support.

This field reads as an IMPLEMENTATION DEFINED value.

**NOTRCPKT, bit [27]**
No trace packets. Indicates whether the implementation does not support trace.

The possible values of this bit are:

**0**
Trace supported.

**1**
Trace not supported.

If this bit is RAZ, the NOCYCCNT bit must also RAZ.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

**NOEXTTRIG, bit [26]**
No External Triggers. Shows whether the implementation does not support external triggers.

Reserved, RES0.

**NOCYCCNT, bit [25]**
No cycle count. Indicates whether the implementation does not include a cycle counter.

The possible values of this bit are:

**0**
Cycle counter implemented.

**1**
Cycle counter not implemented.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

**NOPRFCNT, bit [24]**
No profile counters. Indicates whether the implementation does not include the profiling counters.

The possible values of this bit are:

**0**
Profiling counters implemented.

**1**
Profiling counters not implemented.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

**CYCDISS, bit [23]**
Cycle counter disabled secure. Controls whether the cycle counter is disabled in Secure state.

The possible values of this bit are:

**0**
No effect.

**1**
Disable incrementing of the cycle counter when the PE is in Secure state.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**CYCEVTENA, bit [22]**
Cycle event enable. Enables Event Counter packet generation on POSTCNT underflow.

The possible values of this bit are:

**0**
No Event Counter packets generated when POSTCNT underflows.

**1**
If PCSAMPLENA set to 0, an Event Counter packet is generated when POSTCNT underflows.

RES0 if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**FOLDEVTENA, bit [21]**
Fold event enable. Enables DWT_FOLDCNT counter.

The possible values of this bit are:

**0**
DWT_FOLDCNT disabled.

**1**
DWT_FOLDCNT enabled.

RES0 if the NOPRFCNT bit is RAO. The reset value is 0.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**LSUEVTENA, bit [20]**
LSU event enable. Enables DWT_LSUCNT counter.

The possible values of this bit are:

**0**
DWT_LSUCNT disabled.

**1**
DWT_LSUCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**SLEEPEVTENA, bit [19]**
Sleep event enable. Enable DWT_SLEEPCNT counter.

The possible values of this bit are:

**0**
DWT_SLEEPCNT disabled.

**1**
DWT_SLEEPCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**EXCEVTENA, bit [18]**
Exception event enable. Enables DWT_EXCCNT counter.

The possible values of this bit are:

**0**
DWT_EXCCNT disabled.

**1**
DWT_EXCCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**CPIEVTENA, bit [17]**
CPI event enable. Enables DWT_CPICNT counter.

The possible values of this bit are:

**0**
DWT_CPICNT disabled.

**1**
DWT_CPICNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**EXCTRCENA, bit [16]**
Exception trace enable. Enables generation of Exception Trace packets.

The possible values of this bit are:

**0**
Exception Trace packet generation disabled.

**1**
Exception Trace packet generation enabled.

RES0 if the NOTRCPKT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**Bits [15:13]**
Reserved, RES0.

**PCSAMPLENA, bit [12]**
PC sample enable. Enables use of POSTCNT counter as a timer for Periodic PC Sample packet generation.

The possible values of this bit are:

**0**
Periodic PC Sample packet generation disabled.

**1**
Periodic PC Sample packet generated on POSTCNT underflow.

RES0 if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

**SYNCTAP, bits [11:10]**
Synchronization tap. Selects the position of the synchronization packet request counter tap on the CYCCNT counter. This determines the rate of Synchronization packet requests made by the DWT.

The possible values of this field are:

**0b00**
Synchronization packet request disabled.

**0b01**
Synchronization counter tap at CYCCNT[24].

**0b10**
Synchronization counter tap at CYCCNT[26].

**0b11**
Synchronization counter tap at CYCCNT[28].

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

**CYCTAP, bit [9]**
Cycle count tap. Selects the position of the POSTCNT tap on the CYCCNT counter.

The possible values of this bit are:

**0**
POSTCNT tap at CYCCNT[6].

**1**
POSTCNT tap at CYCCNT[10].

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Cold reset.

**POSTINIT, bits [8:5]**
POSTCNT initial. Initial value for the POSTCNT counter.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

**POSTPRESET, bits [4:1]**
POSTCNT preset. Reload value for the POSTCNT counter.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

**CYCCNTENA, bit [0]**
CYCCNT enable. Enables CYCCNT.

The possible values of this bit are:

**0**
CYCCNT disabled.

**1**
CYCCNT enabled.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

## D1.2.58 DWT_CYCCNT, DWT Cycle Count Register

The DWT_CYCCNT characteristics are:

**Purpose**
Shows or sets the value of the processor cycle counter, CYCCNT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOCYCCNT == 0.

This register is RES0 if DWT_CTRL.NOCYCCNT == 1.

**Attributes**
32-bit read/write register located at `0xE0001004`.

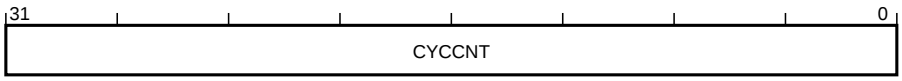
This register is not banked between Security states.

### Preface

The PMU_CCNTR register, if implemented, is an alias of this register.

### Field descriptions

The DWT_CYCCNT bit assignments are:




**CYCCNT, bits [31:0]**
Incrementing cycle counter value. Increments one on each processor clock cycle when DWT_CTRL.CYCCNTENA == 1 and DEMCR.TRCENA == 1. On overflow, CYCCNT wraps to zero.

This field resets to an UNKNOWN value on a Cold reset.

## D1.2.59 DWT_DEVARCH, DWT Device Architecture Register

The DWT_DEVARCH characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

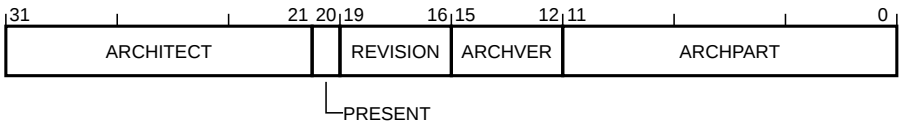
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FBC`.

This register is not banked between Security states.

### Field descriptions

The DWT_DEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**
Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

**`0x23B`**
JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

**PRESENT, bit [20]**
DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

**1**
DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**
Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**
DWT architecture v2.0.

**0b0001**
DWT architecture v2.1. The DWT_VMASKn registers are implemented.

DWT architecture v2.1 is mandatory for a DWT implementation that includes data value comparators in a PE that implements v8.1-M and MVE. If the DWT implementation does not include data value comparators, it is IMPDEF whether it is v2.0 or v2.1.

This field reads as an IMPLEMENTATION DEFINED value.

**ARCHVER, bits [15:12]**
Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**
DWT architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0001`.

**ARCHPART, bits [11:0]**
Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA02**
DWT architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA02`.

## D1.2.60 DWT_DEVTYPE, DWT Device Type Register

The DWT_DEVTYPE characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

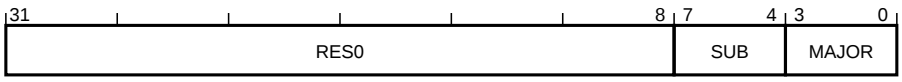
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FCC`.

This register is not banked between Security states.

### Field descriptions

The DWT_DEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Sub-type. Component sub-type.

The possible values of this field are:

**`0x0`**
Other.

This field reads as `0b0000`.

**MAJOR, bits [3:0]**
Major type. Component major type.

The possible values of this field are:

**`0x0`**
Miscellaneous.

This field reads as `0b0000`.

### D1.2.61 DWT_EXCCNT, DWT Exception Overhead Count Register

The DWT_EXCCNT characteristics are:

**Purpose**
Counts the total cycles spent in exception processing.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

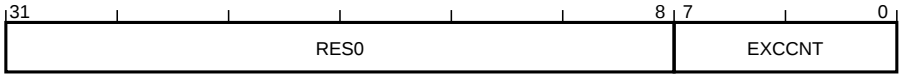
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

**Attributes**
32-bit read/write register located at `0xE000100C`.

This register is not banked between Security states.

## Field descriptions

The DWT_EXCCNT bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**EXCCNT, bits [7:0]**
The exception overhead counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.EXCEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed, see DWT_CPICNT.
- An exception-entry or exception-exit related operation is in progress.
- Either SecureNoninvasiveDebugAllowed() == TRUE, or NS-Req for the operation is set to Non-secure and NoninvasiveDebugAllowed() == TRUE.

Exception-entry or exception-exit related operations include the stacking of registers on exception entry, lazy state preservation, unstacking of registers on exception exit, and preemption.

Initialized to zero when the counter is disabled and DWT_CTRL.EXCEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

### D1.2.62 DWT_FOLDCNT, DWT Folded Instruction Count Register

The DWT_FOLDCNT characteristics are:

**Purpose**
Increments for each additional instruction executed in the current cycle.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

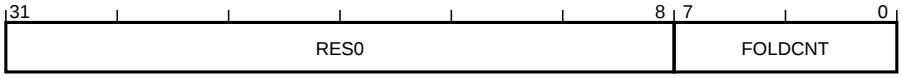
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

**Attributes**
32-bit read/write register located at `0xE0001018`.

This register is not banked between Security states.

## Field descriptions

The DWT_FOLDCNT bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**FOLDCNT, bits [7:0]**
Folded instruction counter.

Counts on each cycle when all of the following are true:

- DWT_CTRL.FOLDEVTENA == 1 and DEMCR.TRCENA == 1.
- At least two instructions are executed, see DWT_CPICNT.
- Either SecureNoninvasiveDebugAllowed() == TRUE, or the PE is in Non-secure state and NoninvasiveDebugAllowed() == TRUE.

The counter is incremented by the number of instructions executed, minus one.

Initialized to zero when the counter is disabled and DWT_CTRL.FOLDEVTENA is written with 1. An event is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

### D1.2.63 DWT_FUNCTIONn, DWT Comparator Function Register, n = 0 - 14

The DWT_FUNCTION{0..14} characteristics are:

**Purpose**
Controls the operation of watchpoint comparator *n*.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the DWT is implemented.

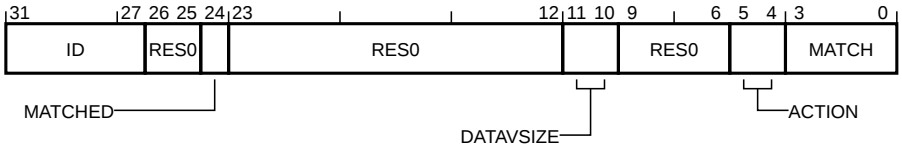
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read/write register located at `0xE0001028` + 16*n*.

This register is not banked between Security states.

## Field descriptions

The DWT_FUNCTION{0..14} bit assignments are:




**ID, bits [31:27]**
Identify capability. Identifies the capabilities for MATCH for comparator *n*.

The possible values of this field are:

**`0b00000`**
Reserved.

**`0b01000`**
Data Address, and Data Address With Value.

**`0b01001`**
Cycle Counter, Data Address, and Data Address With Value.

**`0b01010`**
Instruction Address, Data Address, and Data Address With Value.

**`0b01011`**
Cycle Counter, Instruction Address, Data Address and Data Address With Value.

**`0b11000`**
Data Address, Data Address Limit, and Data Address With Value.

**0b11010**
Instruction Address, Instruction Address Limit, Data Address, Data Address Limit, and Data Address With Value.

**0b11100**
Data Address, Data Address Limit, Data Value, Linked Data Value, and Data Address With Value.

**0b11110**
Instruction Address, Instruction Address Limit, Data Address, Data Address Limit, Data value, Linked Data Value, and Data Address With Value.

All other values are reserved.

Comparator 0 never supports linking. If more than one comparator is implemented, then at least one comparator must support linking. Arm recommends that odd-numbered comparators support linking.

Cycle Counter matching is only supported if the Main Extension is implemented and DWT_CTRL.NOCYCCNT == 0, meaning the cycle counter is implemented. Comparator 0 must support Cycle Counter matching if the cycle counter is implemented.

Data Address With Value is supported for the first four comparators only, and only if the Main Extension and ITM are implemented, and DWT_CTRL.NOTRCPKT == 0. Data Value and Linked Data Value not supported if the Main Extension is not implemented.

This field is read-only.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [26:25]**
Reserved, RES0.

**MATCHED, bit [24]**
Comparator matched. Set to 1 when the comparator matches.

The possible values of this bit are:

**0**
No match.

**1**
Match. The comparator has matched since the last read of this register.

For an Instruction Address Limit or Data Address Limit comparator, this bit is UNKNOWN on reads.

This bit is read-only.

This bit clears to zero when read.

This bit resets to an UNKNOWN value on a Cold reset.

**Bits [23:12]**
Reserved, RES0.

**DATAVSIZE, bits [11:10]**
Data value size. Defines the size of the object being watched for by Data Value and Data Address comparators.

The possible values of this field are:

**0b00**
1 byte.

**0b01**
2 bytes.

**0b10**
4 bytes.

All other values are reserved.

For an Instruction Address or Instruction Address Limit comparator, DATAVSIZE must be `0b01` (2 bytes). If this comparator is part of an data address range pair, DATAVSIZE must be `0b00` (1 byte).

For a Data Address comparator, DWT_COMP*n* must be aligned to the size specified by DATAVSIZE. For a Data Value or Linked Data Value comparator:

- For halfword comparisons, DWT_COMP*n* [31:16] must be equal to DWT_COMP*n*[15:0]. .
- For byte comparisons, DWT_COMP*n* [31:24], DWT_COMP*n* [23:16], and DWT_COMP*n* [15:8] must be equal to DWT_COMP*n* [7:0], and, if implemented, DWT_VMASK*n*[31:24], DWT_VMASK*n*[23:16], DWT_VMASK*n*[15:8] must be equal to DWT_COMP*n* [7:0].

This field resets to an UNKNOWN value on a Cold reset.

**Bits [9:6]**
Reserved, RES0.

**ACTION, bits [5:4]**
Action on match. Defines the action on a match. This field is ignored and the comparator generates no actions if it is disabled by MATCH.

The possible values of this field are:

**`0b00`**
Trigger only.

**`0b01`**
Generate debug event.

**`0b10`**
For a Cycle Counter, Instruction Address, Data Address, Data Value or Linked Data Value comparator, generate a Data Trace Match packet.

For a Data Address With Value comparator, generate a Data Trace Data Value packet.

**`0b11`**
For a Data Address Limit comparator, generate a Data Trace Data Address packet.

For a Cycle Counter, Instruction Address Limit, or Data Address comparator, generate a Data Trace PC Value packet.

For a Data Address With Value comparator, generate both a Data Trace PC Value packet and a Data Trace Data Value packet.

If the Main Extension is not implemented, the values `0b10` and `0b11` are reserved.

This field resets to an UNKNOWN value on a Cold reset.

**MATCH, bits [3:0]**
Match type. Controls the type of match generated by this comparator.

The possible values of this field are:

**`0b0000`**
Disabled. Never generates a match.

**`0b0001`**
Cycle Counter. Matches if DWT_CYCCNT equals the comparator value. The comparator is checked each time DWT_CYCCNT is written to, directly or indirectly.

Only supported if the Main Extension is implemented, DWT_FUNCTION<n>.ID<0> == 1 and DWT_CTRL.NOCYCCNT == 0.

**0b0010**
Instruction Address. If not linked to, an instruction matches if the address of the first byte of the instruction matches the comparator address.

Only supported if DWT_FUNCTION<n>.ID<1> == 1.

**0b0011**
Instruction Address Limit. An instruction matches if the address of the first byte of the instruction lies between the lower comparator address (specified by comparator <*n*-1>) and the limit comparator address (specified by this comparator, <*n*>). Both addresses are inclusive to the range. Comparator <*n*-1> must be programmed for Instruction Address (`0b0010`) or Disabled (`0b0000`), and the lower address must be strictly less-than the limit comparator address, otherwise it is UNPREDICTABLE whether or not any comparator generates matches.

Only supported if DWT_FUNCTION<n>.ID<4> == 1 and DWT_FUNCTION<n>.ID<1> == 1.

**0b0100**
Data Address. If not linked to by a Data Address Limit comparator, an access matches if any accessed byte lies between the comparator address and a limit defined by the DATAVSIZE field. Supported for all comparators.

**0b0101**
Data Address, writes. As `0b0100`, except that only write accesses generate a match.

**0b0110**
Data Address, reads. As `0b0100`, except that only read accesses generate a match.

**0b0111**
Data Address Limit. An access matches if any byte made by the access lies between the lower address (specified by comparator <*n*-1>) and the limit address (specified by this comparator, <*n*>). Both addresses are inclusive to the range. Comparator <*n*-1> must be programmed for Data Address (`0b01xx`, not `0b0111`), Data Address With Data Value (`0b11xx`, not `0b1111`), or Disabled (`0b0000`), and the lower address must be strictly less-than the limit comparator address, otherwise it is UNPREDICTABLE whether or not any comparator generates matches. DWT_FUNCTION<*n*-1>.MATCH[1:0] determines the matching access types.

Only supported if DWT_FUNCTION<n>.ID<4> == 1.

**0b1000**
Data Value. An access matches if the value accessed matches the comparator value.

Only supported if the Main Extension is implemented and DWT_FUNCTION<n>.ID<2> == 1.

**0b1001**
Data Value, writes. As `0b1000`, except that only write accesses generate a match.

**0b1010**
Data Value, reads. As `0b1000`, except that only read accesses generate a match.

**0b1011**
Linked Data Value. An access matches if the value accessed matches the comparator value (specified by comparator <*n*>) and the linked data address (specified by comparator <*n*-1>) for the same access matches. Comparator <*n*-1> must be programmed for Data Address (`0b01xx`, not `0b0111`), or Data Address With Value (`0b11xx`, not `0b1111`), or Disabled (`0b0000`), and DATAVSIZE for the two comparators must be the same, otherwise it is UNPREDICTABLE whether or not any comparator generates matches. DWT_FUNCTION<*n*-1>.MATCH[1:0] determines the matching access types.

Only supported if the Main Extension is implemented and DWT_FUNCTION<n>.ID<4> == 1 and DWT_FUNCTION<n>.ID<2> == 1.

**0b1100**
Data Address With Value. As `0b01xx`, except that the data value is traced.

Supported for the first four comparators only, and only if DWT_CTRL.NOTRCPKT == 0 and ITM is also implemented.

**0b1101**
Data Address With Value, writes. As `0b1100`, except that only write accesses generate a match.

**0b1110**
Data Address With Value, reads. As `0b1100`, except that only read accesses generate a match.

Any value not supported by the comparator is reserved. For a pair of linked comparators, <*n*> and <*n*-1>, DWT_FUNCTION<*n*-1>.MATCH[1:0] determines the matching access types. See MATCH table for further details.

This field resets to zero on a Cold reset.

### D1.2.64 DWT_LAR, DWT Software Lock Access Register

The DWT_LAR characteristics are:

**Purpose**

Provides CoreSight Software Lock control for the DWT, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

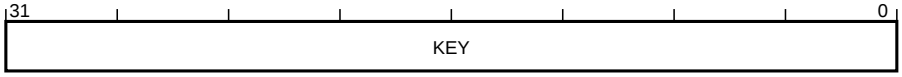
**Attributes**

32-bit write-only register located at `0xE0001FB0`.

This register is not banked between Security states.

## Field descriptions

The DWT_LAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.65 DWT_LSR, DWT Software Lock Status Register

The DWT_LSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the DWT, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

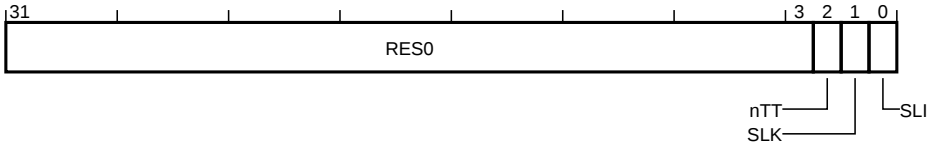
**Attributes**

32-bit read-only register located at `0xE0001FB4`.

This register is not banked between Security states.

## Field descriptions

The DWT_LSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

### D1.2.66 DWT_LSUCNT, DWT LSU Count Register

The DWT_LSUCNT characteristics are:

**Purpose**
Increments on the additional cycles required to execute all load or store instructions.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

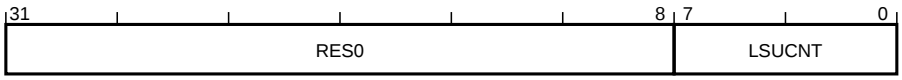
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

**Attributes**
32-bit read/write register located at `0xE0001014`.

This register is not banked between Security states.

## Field descriptions

The DWT_LSUCNT bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**LSUCNT, bits [7:0]**
Load-store overhead counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.LSUEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed, see DWT_CPICNT.
- No exception-entry or exception-exit operation is in progress, see DWT_EXCCNT.
- A load-store operation is in progress.
- Either SecureNoninvasiveDebugAllowed() == TRUE, or NS-Req for the operation is set to Non-secure and NoninvasiveDebugAllowed() == TRUE.

Initialized to zero when the counter is disabled and DWT_CTRL.LSUEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

### D1.2.67 DWT_PCSR, DWT Program Counter Sample Register

The DWT_PCSR characteristics are:

**Purpose**
Samples the current value of the Program Counter.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the DWT is implemented.

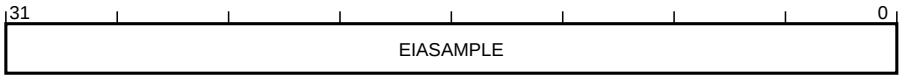
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE000101C`.

This register is not banked between Security states.

## Field descriptions

The DWT_PCSR bit assignments are:




**EIASAMPLE, bits [31:0]**
Executed instruction address sample. Recently executed instruction address sample value.

The possible values of this field are:

**`0xFFFFFFFF`**
One of the following is true:

- The PE is halted in Debug state.
- The Security Extension is implemented, the sampled instruction was executed in Secure state, and SecureNoninvasiveDebugAllowed() == FALSE.
- NoninvasiveDebugAllowed() == FALSE.
- DEMCR.TRCENA == 0.
- The address of a recently-executed instruction is not available.

**Not `0xFFFFFFFF`**
Instruction address of a recently executed instruction. Bit [0] of the sample instruction address is 0.

The conditions when the address of a recently-executed instruction is not available are IMPLEMENTATION DEFINED.

### D1.2.68 DWT_PIDR0, DWT Peripheral Identification Register 0

The DWT_PIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

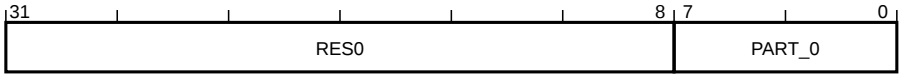
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FE0`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.69 DWT_PIDR1, DWT Peripheral Identification Register 1

The DWT_PIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

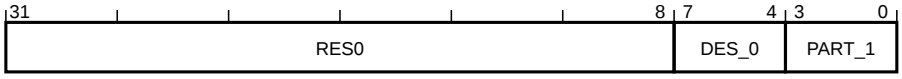
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FE4`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.70 DWT_PIDR2, DWT Peripheral Identification Register 2

The DWT_PIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

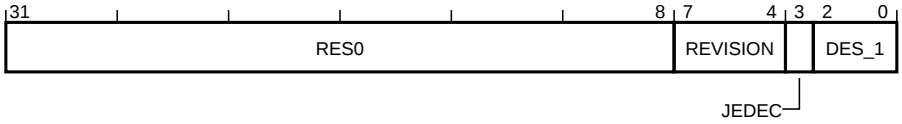
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FE8`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

**DES_1, bits [2:0]**
JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.71 DWT_PIDR3, DWT Peripheral Identification Register 3

The DWT_PIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

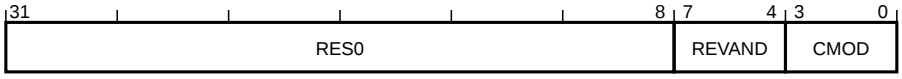
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FEC`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.72 DWT_PIDR4, DWT Peripheral Identification Register 4

The DWT_PIDR4 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

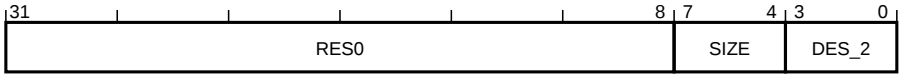
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FD0`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

**DES_2, bits [3:0]**
JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.73 DWT_PIDR5, DWT Peripheral Identification Register 5

The DWT_PIDR5 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

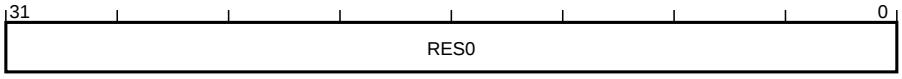
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FD4`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.74 DWT_PIDR6, DWT Peripheral Identification Register 6

The DWT_PIDR6 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

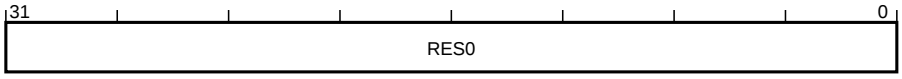
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FD8`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR6 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.75 DWT_PIDR7, DWT Peripheral Identification Register 7

The DWT_PIDR7 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the DWT.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

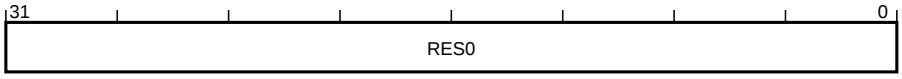
This register is RES0 if the DWT is not implemented.

**Attributes**
32-bit read-only register located at `0xE0001FDC`.

This register is not banked between Security states.

## Field descriptions

The DWT_PIDR7 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.76 DWT_SLEEPCNT, DWT Sleep Count Register

The DWT_SLEEPCNT characteristics are:

**Purpose**
Counts the total number of cycles that the processor is sleeping.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

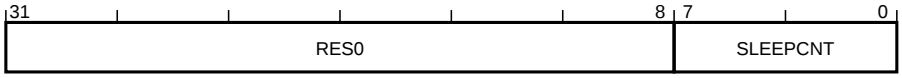
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

**Attributes**
32-bit read/write register located at `0xE0001010`.

This register is not banked between Security states.

## Field descriptions

The DWT_SLEEPCNT bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SLEEPCNT, bits [7:0]**
Sleep counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.SLEEPEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed, see DWT_CPICNT.
- No load-store operation is in progress, see DWT_LSUCNT.
- No exception-entry or exception-exit operation is in progress, see DWT_EXCCNT.
- The PE is in a power-saving mode.
- Either SecureNoninvasiveDebugAllowed() == TRUE, or the PE is in Non-secure state and NoninvasiveDebugAllowed() == TRUE.

Power-saving modes include WFI, WFE, and Sleep-on-exit.

All power-saving features are IMPLEMENTATION DEFINED and therefore when this counter counts is IMPLEMENTATION DEFINED. In particular, it is IMPLEMENTATION DEFINED whether the counter increments if the PE is in a power-saving mode and SCR.SLEEPDEEP is set.

Initialized to zero when the counter is disabled and DWT_CTRL.SLEEPEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

**Note**

Arm recommends that this counter counts all cycles when the PE is sleeping and SCR.SLEEPDEEP is clear, regardless of whether a WFI or WFE instruction, or Sleep-on-exit, caused the entry to the power-saving mode.

### D1.2.77 DWT_VMASKn, DWT Comparator Value Mask Register, n = 0 - 14

The DWT_VMASK{0..14} characteristics are:

**Purpose**
Provides a mask value for use by watchpoint comparator *n* when comparing data values.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if the DWT implements DWT architecture version 2.1 or later.

This register is RES0 if the DWT implements DWT architecture version 2.0 or earlier.

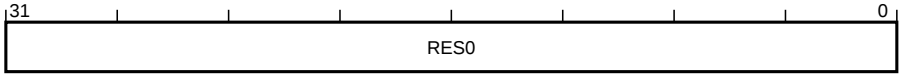
**Attributes**
32-bit read/write register located at `0xE000102C` + 16*n*.

This register is not banked between Security states.

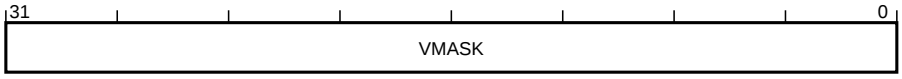
## Field descriptions

The DWT_VMASK{0..14} bit assignments are:

**When DWT_FUNCTIONn.MATCH != 0b10xx**:




**When DWT_FUNCTIONn.MATCH == 0b10xx**:




**Bits [31:0], when DWT_FUNCTIONn.MATCH != 0b10xx**
Reserved, RES0.

**VMASK, bits [31:0], when DWT_FUNCTIONn.MATCH == 0b10xx**
Data value mask. Mask value for use in the comparison with load or store data.

The possible values of each bit are:

**0**
The comparison matches only if DWT_COMPn[*m*] matches bit [*m*] of the candidate data value.

**1**
The comparison ignores bit [*m*] of the candidate data value. If DWT_COMPn[*m*] is not set to zero, the result of the comparison is UNPREDICTABLE.

For halfword or word comparisons, the mask is in little-endian order. That is, the least significant byte of this register masks the byte targeting the lowest address in memory.

For byte or halfword comparisons, if the value of the byte or halfword is not replicated across all byte or halfword lanes, the value used for the comparison is UNKNOWN.

This field resets to an UNKNOWN value on a Cold reset.

## D1.2.78 EPSR, Execution Program Status Register

The EPSR characteristics are:

**Purpose**
Holds Execution state bits.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**
This register is always implemented.

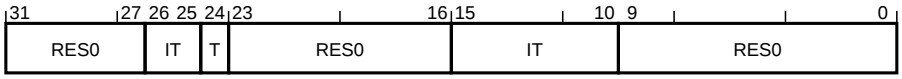
**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

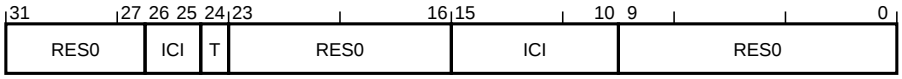
### Field descriptions

The EPSR bit assignments are:

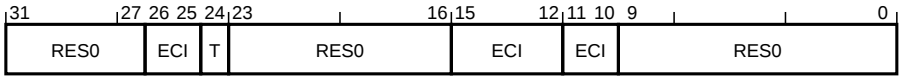
**When {EPSR[26:25], EPSR[11:10]} != 0**:




**When {EPSR[26:25], EPSR[11:10]} == 0, and a multicycle load or store instruction is in progress**:




**When {EPSR[26:25], EPSR[11:10]} == 0, and beat-wise vector instructions are in progress**:




**Bits [31:27]**
Reserved, RES0.

**T, bit [24]**
T32 state bit. Determines the current instruction set state.

The possible values of this bit are:

**0**
Execution of any instruction generates an INVSTATE UsageFault.

**1**
Instructions decoded as T32 instructions.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [23:16]**
Reserved, RES0.

**IT, bits [15:10, 26:25]** , when [{EPSR[26:25], EPSR[11:10]} != 0]
If-then flags. This field encodes the current condition and position in an IT block sequence.

The field IT[7:0] is equivalent to EPSR[15:10,26:25].

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

**ICI, bits [26:25, 15:10]** , when [{EPSR[26:25], EPSR[11:10]} == 0, and a multicycle load or store instruction is in progress]
Interrupt continuation flags. This field encodes information on the outstanding register list for an interrupted exception-continuable multicycle load or store instruction.

The field ICI[7:0] is equivalent to EPSR[26:25,15:10].

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

**ECI, bits [26:25, 11:10, 15:12]** , when [{EPSR[26:25], EPSR[11:10]} == 0, and beat-wise vector instructions are in progress]
Exception continuation flags for beat-wise vector instructions. This field encodes which beats of the in-flight instructions have completed.

The possible values of this field are:

**0b00000000**
No completed beats.

**0b00000001**
Completed beats: A0.

**0b00000010**
Completed beats: A0 A1.

**0b00000011**
Reserved.

**0b00000100**
Completed beats: A0 A1 A2.

**0b00000101**
Completed beats: A0 A1 A2 B0.

**0b0000011X**
Reserved.

**0b00001XXX**
Reserved.

In the enumeration above the letters correspond to the instructions at the return address and beyond, whilst the numbers correspond to the beats of those instructions that have been completed. For example, the sequence A0 A1 A2 B0 means the first three beats of the instruction at the return address, plus the first beat of the instruction at the return address +4 have been completed. The field ECI[7:0] is equivalent to EPSR[26:25,11:10,15:12].

This field resets to zero on a Warm reset.

**Bits [9:0]**
Reserved, RES0.

## D1.2.79 ERRADDRn, Error Record Address Register, n = 0 - 55

The ERRADDR{0..55} characteristics are:

**Purpose**

If an error has an associated address, this must be written to the address register when the error is recorded. It is IMPLEMENTATION DEFINED how the recorded addresses map to the software-visible physical addresses. Software might have to reconstruct the actual physical addresses using the identity of the node and knowledge of the system. Ignores writes if ERR<*n*>STATUS.AV is set to 1.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005018` + 64*n*.

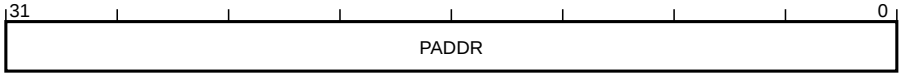
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Field descriptions

The ERRADDR{0..55} bit assignments are:




**PADDR, bits [31:0]**

Address, bits [31:0]. Unimplemented bits are RES0.

## D1.2.80 ERRADDR2n, Error Record Address 2 Register, n = 0 - 55

The ERRADDR2{0..55} characteristics are:

**Purpose**

If an error has an associated address, this must be written to the address register when the error is recorded. It is IMPLEMENTATION DEFINED how the recorded addresses map to the software-visible physical addresses. Software might have to reconstruct the actual physical addresses using the identity of the node and knowledge of the system. Ignores writes if ERR<*n*>STATUS.AV is set to 1.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE000501C` + 64*n*.

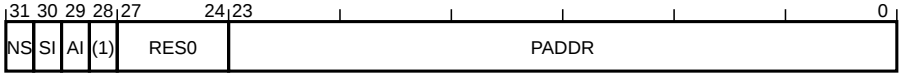
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Field descriptions

The ERRADDR2{0..55} bit assignments are:




**NS, bit [31]**

Non-secure attribute.

The possible values of this bit are:

**`0b0`**

The address is Secure.

**`0b1`**

The address is Non-secure.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

**SI, bit [30]**

Secure Incorrect. Indicates whether the NS bit is valid.

The possible values of this bit are:

**`0b0`**

The NS bit is correct. That is, it matches the programmers' view of the Non-secure attribute for this recorded location.

**`0b1`**

The NS bit might not be correct, and might not match the programmers' view of the Non-secure attribute for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

**AI, bit [29]**
Address Incomplete or incorrect. Indicates whether the PADDR field in ERR<*n*>ADDR and ERR<*n*>ADDR2 is a valid physical address.

The possible values of this bit are:

**0b0**
The PADDR field is a valid physical address. That is, it matches the programmers' view of the physical address for this recorded location.

**0b1**
The PADDR field might not be a valid physical address, and might not match the programmers' view of the physical address for the recorded location.

It is IMPLEMENTATION DEFINED whether this bit is read-only or read/write.

**Bit [28]**
This bit reads as zero.

**Bits [27:24]**
Reserved, RES0.

**PADDR, bits [23:0]**
Address, bits [55:32]. Unimplemented bits are RES0.

## D1.2.81 ERRCTRLn, Error Record Control Register, n = 0 - 55

The ERRCTRL{0..55} characteristics are:

**Purpose**
The error control register contains enable bits for the node that writes to this record.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**
32-bit read/write register located at `0xE0005008` + 64*n*.

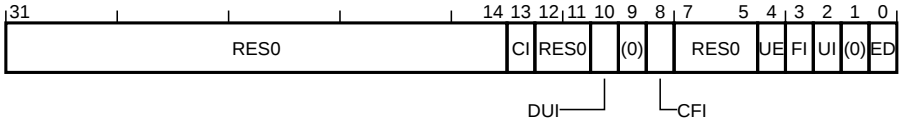
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Field descriptions

The ERRCTRL{0..55} bit assignments are:




**Bits [31:14]**
Reserved, RES0.

**CI, bit [13]**
Critical error interrupt enable. When enabled the critical error interrupt is generated for a critical error condition.

The possible values of this bit are:

**`0b0`**
Critical error interrupt not generated for critical errors. Critical errors are treated as Uncontained errors.

**`0b1`**
Critical error interrupt generated for critical errors.

This bit is RES0 if the node does not support this control.

**Bits [12:11]**
Reserved, RES0.

**DUI, bit [10]**
Enable error recovery interrupt enable for deferred errors. When enabled the error recovery interrupt is generated for all detected Deferred errors.

The possible values of this bit are:

**0b0**
Error recovery interrupt not generated for deferred errors.

**0b1**
Error recovery interrupt generated for deferred errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error. This bit is RES0 if the node does not support this control.

**Bit [9]**
Reserved, RES0.

**CFI, bit [8]**
Enable fault handling interrupt for corrected errors. When enabled, if the node implements a Corrected error counter, then the fault handling interrupt is generated when the counter overflows and the overflow bit is set. Otherwise the fault handling interrupt is also generated for all detected Corrected errors.

The possible values of this bit are:

**0b0**
Fault handling interrupt not generated for corrected errors.

**0b1**
Fault handling interrupt generated for corrected errors.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error. This bit is RES0 if the node does not support this control.

**Bits [7:5]**
Reserved, RES0.

**UE, bit [4]**
Enable in-band uncorrected error reporting. When enabled, responses to transactions that detect an uncorrected error that cannot be deferred are signaled as a detected error (external abort).

The possible values of this bit are:

**0b0**
External abort response for uncorrected errors disabled.

**0b1**
External abort response for uncorrected errors enabled.

This bit is RES0 if the node does not support this control.

**FI, bit [3]**
Enable fault handling interrupt. When enabled, the fault handling interrupt is generated for all detected Deferred errors and Uncorrected errors. If the fault handling interrupt for corrected errors control is not implemented then if the node implements a Corrected error counter then the fault handling interrupt is also generated when the counter overflows and the overflow bit is set, otherwise the fault handling interrupt is also generated for all detected Corrected errors.

The possible values of this bit are:

**0b0**
Fault handling interrupt disabled.

**0b1**
Fault handling interrupt enabled.

This bit is RES0 if the node does not support this control.

**UI, bit [2]**
Enable error recovery interrupt. Uncorrected error recovery interrupt enable. When enabled, the error recovery interrupt is generated for all detected Uncorrected errors that are not deferred.

The possible values of this bit are:

**0b0**
Error recovery interrupt disabled.

**0b1**
Error recovery interrupt enabled.

The interrupt is generated even if the error syndrome is discarded because the error record already records a higher priority error. This bit is RES0 if the node does not support this control.

**Bit [1]**
Reserved for IMPLEMENTATION DEFINED controls. Must permit SBZP write policy for software. This bit reads as an IMPLEMENTATION DEFINED value and writes to this bit have IMPLEMENTATION DEFINED behavior.

Reserved, RES0.

**ED, bit [0]**
Error reporting and logging enable. When disabled, the node behaves as if error detection and correction are disabled, and no errors are recorded or signaled by the node. ARM recommends that, when disabled, correct error detection and correction codes are written for writes, unless disabled by an IMPLEMENTATION DEFINED control for error injection.

The possible values of this bit are:

**0b0**
Error reporting disabled.

**0b1**
Error reporting enabled.

It is IMPLEMENTATION DEFINED whether the node fully disables error detection and correction when reporting is disabled. That is, even with error reporting disabled, the node might continue to silently correct errors. Uncorrectable errors might result in corrupt data being silently propagated by the node. This bit is RES1 if the node does not support this control.

This bit resets to zero on a Cold reset.

## D1.2.82 ERRDEVID, Error Record Device ID Register

The ERRDEVID characteristics are:

**Purpose**
Defines the number of error records that can be accessed through the Memory-mapped registers.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if RAS is implemented.

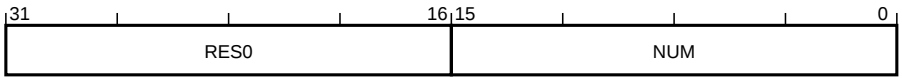
This register is RES0 if RAS is not implemented.

**Attributes**
32-bit read-only register located at `0xE0005FC8`.

This register is not banked between Security states.

### Field descriptions

The ERRDEVID bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**NUM, bits [15:0]**
Number of implemented error record indexes. Indicates the highest numbered index of the records that can be accessed, plus one. Zero indicates that no records can be accessed. Each record is notionally owned by a node, though a node might implement multiple records.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.83 ERRFRn, Error Record Feature Register, n = 0 - 55

The ERRFR{0..55} characteristics are:

**Purpose**

Identifies the features implemented by the associated record *n*, and of those features that are software programmable.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read-only register located at `0xE0005000` + 64*n*.

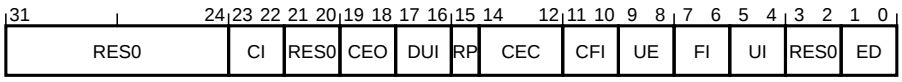
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Field descriptions

The ERRFR{0..55} bit assignments are:




**Bits [31:24]**

Reserved, RES0.

**CI, bits [23:22]**

Critical Error Interrupt. Indicates whether the critical error interrupt and associated controls are implemented.

The possible values of this field are:

**`0b00`**

Does not support feature.

**`0b01`**

Feature always enabled.

**`0b10`**

Feature is controllable.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [21:20]**

Reserved, RES0.

**CEO, bits [19:18]**

Corrected Error overwrite. Indicates the behavior when a second Corrected error is detected after a first Corrected error has been recorded by the node.

The possible values of this field are:

**0b00**
Count Corrected error if a counter is implemented. Keep the previous error syndrome. If the counter overflows, or no counter is implemented then ERR<*n*>STATUS.OF is set to 1.

**0b01**
Count Corrected error. If ERR<*n*>STATUS.OF == 1 before the Corrected error is counted then keep the previous syndrome. Otherwise the previous syndrome is overwritten. If the counter overflows then ERR<*n*>STATUS.OF is set to 1.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**DUI, bits [17:16]**
Error recovery interrupt for deferred errors. Indicates whether the node implements a control for enabling error recovery interrupts on deferred errors.

The possible values of this field are:

**0b00**
Does not support feature. ERRCTLRn.DUI is RES0.

**0b01**
Feature is controllable using ERRCTLRn.DUI.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**RP, bit [15]**
Repeat counter. Indicates whether the node implements a repeat Corrected error counter in ERR<*n*>MISC0.

The possible values of this bit are:

**0b0**
A single CE counter is implemented.

**0b1**
A first (repeat) counter and a second (other) counter are implemented. The repeat counter is the same size as the primary error counter.

This bit is RES0 if ERR<*n*>FR.CEC == 0b000.

This bit reads as an IMPLEMENTATION DEFINED value.

**CEC, bits [14:12]**
Corrected Error Counter. Indicates whether the node implements a standard Corrected error (CE) counter mechanism in ERR<*n*>MISC0.

The possible values of this field are:

**0b000**
Does not implement the standard Corrected error counter model.

**0b010**
Implements an 8-bit Corrected error counter in ERR<*n*>MISC0[7:0].

**0b100**
Implements a 16-bit Corrected error counter in ERR<*n*>MISC0[15:0].

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**CFI, bits [11:10]**
Fault handling for Corrected errors. Indicates whether the node implements a control for enabling fault handling interrupts on Corrected errors.

The possible values of this field are:

**0b00**
Does not support feature, ERRCTLRn.CFI is RES0.

**0b10**
Feature is controllable using ERRCTLRn.CFI.

All other values are reserved.

This bit is 'RES0 if ERR<*n*>FR.FI == 0b00.

This field reads as an IMPLEMENTATION DEFINED value.

**UE, bits [9:8]**
In-band uncorrected error reporting. Indicates whether the node implements in-band uncorrected error reporting (external aborts), and, if so, whether it implements controls for enabling and disabling in-band uncorrected error reporting.

The possible values of this field are:

**0b00**
Does not support feature, ERRCTLRn.UE is RES0.

**0b01**
Feature always enabled, ERRCTLRn.UE is RES0.

**0b10**
Feature is controllable using ERRCTLRn.UE.

This field reads as an IMPLEMENTATION DEFINED value.

**FI, bits [7:6]**
Fault handling interrupt. Indicates whether the node implements a fault handling interrupt, and, if so, whether it implements controls for enabling and disabling the fault handling interrupt.

The possible values of this field are:

**0b00**
Does not support feature, ERRCTLRn.FI is RES0.

**0b01**
Feature always enabled, ERRCTLRn.FI is RES0.

**0b10**
Feature is controllable using ERRCTLRn.FI.

This field reads as an IMPLEMENTATION DEFINED value.

**UI, bits [5:4]**
Error recovery interrupt for uncorrected errors. Indicates whether the node implements an error recovery interrupt, and, if so, whether it implements controls for enabling and disabling the error recovery interrupt.

The possible values of this field are:

**0b00**
Does not support feature, ERRCTLRn.UI is RES0.

**0b01**
Feature always enabled, ERRCTLRn.UI is RES0.

**0b10**
Feature is controllable using ERRCTLRn.UI.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [3:2]**
Reserved, RES0.

**ED, bits [1:0]**
Error reporting and logging. Indicates whether the node implements controls for enabling and disabling error reporting and logging.

The possible values of this field are:

**0b01**
Feature always enabled, ERRCTLRn.ED is RES0.

**0b10**
Feature is controllable using ERRCTLRn.ED.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.84 ERRGSRn, RAS Fault Group Status Register

The ERRGSR{0..1} characteristics are:

**Purpose**

Each ERR<*n*>GSR register shows the status for up to 32 records in the group. <n> selects the set of 32 records from the records in the group.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

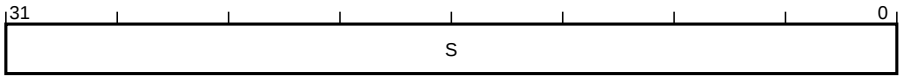
32-bit read-only register located at `0xE0005E00` + 4*n*.

This register is not banked between Security states.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

## Field descriptions

The ERRGSR{0..1} bit assignments are:




**S, bits [31:0]**

Status for Error Record <m>. Each bit is a read-only copy of the corresponding ERR<m>STATUS.V field.

The possible values of this field are:

**`0b0`**

No error.

**`0b1`**

One or more errors.

This bit is RES0 if <m> is greater than or equal to the number of implemented records, or if the record does not support this type of reporting.

## D1.2.85 ERRIIDR, Error Implementer ID Register

The ERRIIDR characteristics are:

**Purpose**
Defines the implementer of the component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if RAS is implemented.

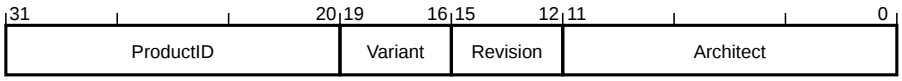
This register is RES0 if RAS is not implemented.

**Attributes**
32-bit read-only register located at `0xE0005E10`.

This register is not banked between Security states.

### Field descriptions

The ERRIIDR bit assignments are:




**ProductID, bits [31:20]**
This field reads as an IMPLEMENTATION DEFINED value.

**Variant, bits [19:16]**
Component Major Revision. This field distinguishes between variants or major revisions of the product.

This field reads as an IMPLEMENTATION DEFINED value.

**Revision, bits [15:12]**
Component minor revision. This field distinguishes between minor revisions of the product.

This field reads as an IMPLEMENTATION DEFINED value.

**Architect, bits [11:0]**
The possible values of this field are:

**`0x23B`**
JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

## D1.2.86 ERRMISC0n, Error Record Miscellaneous 0 Register, n = 0 - 55

The ERRMISC0{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005020` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

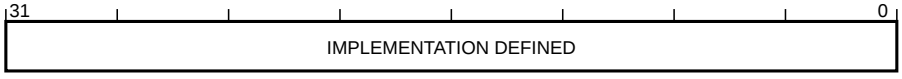
If the node supports the architecturally-defined error counter then it is implemented in ERR<*n*>MISC0.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC0 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

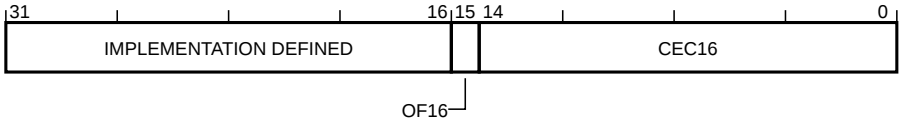
### Field descriptions

The ERRMISC0{0..55} bit assignments are:

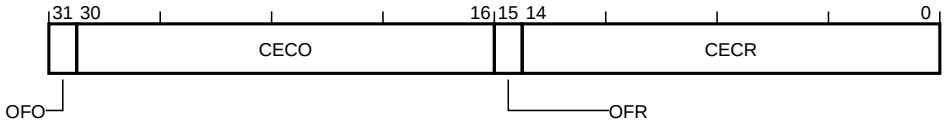
**When Contents are IMPLEMENTATION DEFINED**:



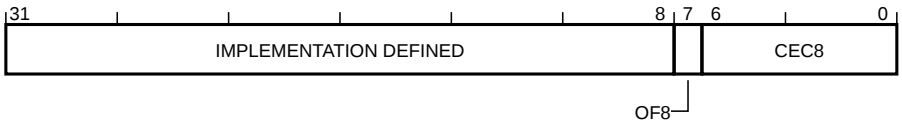

**When Standard 16-bit CE counter**:



**When Standard 16-bit CE counter pair**:




**When Standard 8-bit CE counter**:




**Bits [31:0], when Contents are IMPLEMENTATION DEFINED**
IMPLEMENTATION DEFINED.

**Bits [31:8], when Standard 8-bit CE counter**
IMPLEMENTATION DEFINED.

Reserved, RES0.

**OF8, bit [7], when Standard 8-bit CE counter**
Overflow. Sticky overflow bit.

The possible values of this bit are:

**0b0**
Counter has not overflowed.

**0b1**
Counter has overflowed.

The fault handling interrupt is generated when the corrected fault handling interrupt is enabled and the overflow bit is set to 1.

A direct write that modifies this bit might indirectly set ERR<*n*>STATUS.OF to an UNKNOWN value and a direct write to ERR<*n*>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

**CEC8, bits [6:0], when Standard 8-bit CE counter**
Corrected error count.

**Bits [31:16], when Standard 16-bit CE counter**
IMPLEMENTATION DEFINED.

Reserved, RES0.

**OF16, bit [15], when Standard 16-bit CE counter**
Overflow. Sticky overflow bit.

The possible values of this bit are:

**0b0**
Counter has not overflowed.

**0b1**
Counter has overflowed.

The fault handling interrupt is generated when the corrected fault handling interrupt is enabled and the overflow bit is set to 1.

A direct write that modifies this bit might indirectly set ERR<*n*>STATUS.OF to an UNKNOWN value and a direct write to ERR<*n*>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

**CEC16, bits [14:0], when Standard 16-bit CE counter**
Corrected error count.

**OFO, bit [31], when Standard 16-bit CE counter pair**
Overflow Other. Sticky overflow bit, other.

The possible values of this bit are:

**0b0**
Other Counter has not overflowed.

**0b1**
Other Counter has overflowed.

The fault handling interrupt is generated when the corrected fault handling interrupt is enabled and the overflow bit is set to 1.

A direct write that modifies this bit might indirectly set ERR<*n*>STATUS.OF to an UNKNOWN value and a direct write to ERR<*n*>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

**CECO, bits [30:16], when Standard 16-bit CE counter pair**
Corrected error count, other. Incremented for each Corrected error that is not accounted for by incrementing CECR.

**OFR, bit [15], when Standard 16-bit CE counter pair**
Overflow Repeat. Sticky overflow bit, repeat.

The possible values of this bit are:

**0b0**
Repeat Counter has not overflowed.

**0b1**
Repeat Counter has overflowed.

The fault handling interrupt is generated when the corrected fault handling interrupt is enabled and the overflow bit is set to 1.

A direct write that modifies this bit might indirectly set ERR<*n*>STATUS.OF to an UNKNOWN value and a direct write to ERR<*n*>STATUS.OF that clears it to zero might indirectly set this bit to an UNKNOWN value.

**CECR, bits [14:0], when Standard 16-bit CE counter pair**
Corrected error count. Incremented for the first detected Corrected error, which also records other syndrome for the error, and subsequently for each Corrected error that matches the recorded other syndrome.

## D1.2.87 ERRMISC1n, Error Record Miscellaneous 1 Register, n = 0 - 55

The ERRMISC1{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005024` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

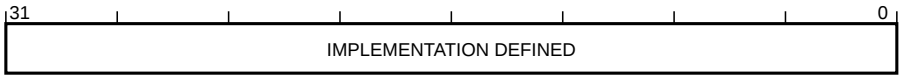
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC1 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC1{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.88 ERRMISC2n, Error Record Miscellaneous 2 Register, n = 0 - 55

The ERRMISC2{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005028` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

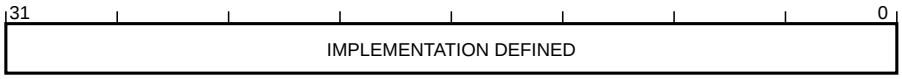
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC2 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC2{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.89 ERRMISC3n, Error Record Miscellaneous 3 Register, n = 0 - 55

The ERRMISC3{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE000502C` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

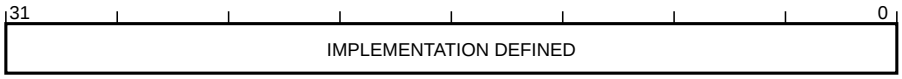
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC3 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC3{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.90 ERRMISC4n, Error Record Miscellaneous 4 Register, n = 0 - 55

The ERRMISC4{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005030` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

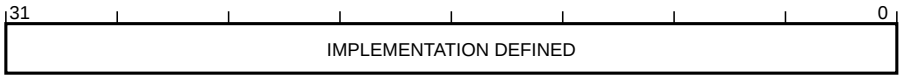
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC4 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC4{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.91 ERRMISC5n, Error Record Miscellaneous 5 Register, n = 0 - 55

The ERRMISC5{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005034` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

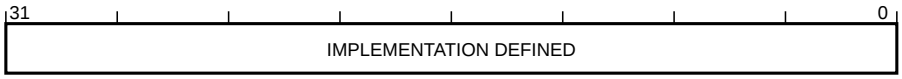
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC5 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC5{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.92 ERRMISC6n, Error Record Miscellaneous 6 Register, n = 0 - 55

The ERRMISC6{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE0005038` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

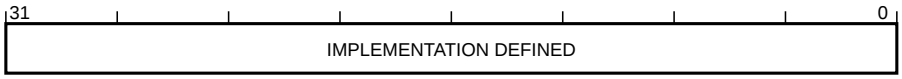
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC6 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC6{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

## D1.2.93 ERRMISC7n, Error Record Miscellaneous 7 Register, n = 0 - 55

The ERRMISC7{0..55} characteristics are:

**Purpose**

IMPLEMENTATION DEFINED error syndrome register.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**

32-bit read/write register located at `0xE000503C` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Preface

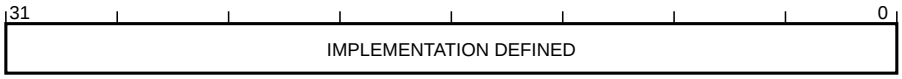
The miscellaneous syndrome registers contain:

- Corrected error counter or counters, if the node supports the counting of Corrected errors.
- Information to identify the Field Replaceable Unit (FRU) in which the error was detected, and might contain enough information to locate error within that FRU.
- Other state information not present in the corresponding status and address registers.

If ERR<*n*>STATUS.MV is set to 1 then it is IMPLEMENTATION DEFINED whether fields of ERR<*n*>MISC7 ignores writes. Arm recommends that miscellaneous syndrome for multiple errors, such as a corrected error counter, is read/write. This allows a counter to be reset in the presence of a persistent error. Miscellaneous syndrome for the most recently recorded error, such as information locating a FRU for that error, should ignore writes. This prevents information being lost if an error is detected whilst the previous error is being logged.

### Field descriptions

The ERRMISC7{0..55} bit assignments are:




**Bits [31:0]**

IMPLEMENTATION DEFINED.

### D1.2.94 ERRSTATUSn, Error Record Primary Status Register, n = 0 - 55

The ERRSTATUS{0..55} characteristics are:

**Purpose**
Contains status information for the error record.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**
32-bit read/write register located at `0xE0005010` + 64*n*.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

## Preface

This register contains the following information:

- Whether any error has been detected (valid).
- Whether any detected error was not corrected, and returned to a master.
- Whether any detected error was not corrected and deferred.
- Whether an error record has been discarded because additional errors have been detected before the first error was handled by software (overflow).
- Whether any error has been reported.
- Whether the other error record registers contain valid information.
- Whether the error was reported because poison data was detected or because a corrupt value was detected by an error detection code.
- A primary error code.
- An IMPLEMENTATION DEFINED extended error code.

Within this register:

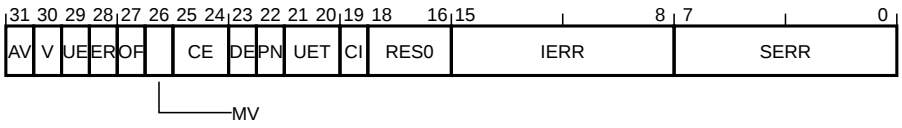
- The {AV, V, MV} bits are valid bits that define whether the error record registers are valid.
- The {UE, OF, CE, DE, UET} bits encode the type of error or errors recorded.
- The {CI, ER, PN, IERR, SERR} fields are syndrome fields.

After reading the status register, software must clear the valid bits to allow new errors to be recorded. Between reading the register and clearing the valid bits, a new error might have overwritten the register. To prevent this error being lost, a write to ERR<*n*>STATUS is ignored if all of:

- Any of the ERR<*n*>STATUS.{V, UE, OF, CE, DE} fields are nonzero before the write.
- The write does not clear the nonzero ERR<*n*>STATUS.{V, UE, OF, CE, DE} field(s) to zero by writing one(s) to the applicable field(s).

## Field descriptions

The ERRSTATUS{0..55} bit assignments are:




**AV, bit [31]**
Address Valid.

The possible values of this bit are:

**0b0**
ERR<*n*>ADDR and ERR<*n*>ADDR2 not valid.

**0b1**
ERR<*n*>ADDR and ERR<*n*>ADDR2 contain an address associated with the highest priority error recorded by this record.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset.

**V, bit [30]**
Status Register valid.

The possible values of this bit are:

**0b0**
ERR<*n*>STATUS not valid.

**0b1**
ERR<*n*>STATUS valid. At least one error has been recorded.

This bit is read/write-one-to-clear.

This bit resets to zero on a Cold reset.

**UE, bit [29]**
Uncorrected error or errors.

The possible values of this bit are:

**0b0**
No errors that could neither be corrected nor deferred.

**0b1**
At least one error that has neither been corrected nor deferred.

This bit reads as UNKNOWN if ERR<*n*>STATUS.V is set to 0. This bit is read/write-one-to-clear.

**ER, bit [28]**
Error Reported.

The possible values of this bit are:

**0b0**
No in-band error (external abort) reported.

**0b1**
An external abort was signaled by the node to the master making the access or other transaction. This can be because any of:

- ERR<n>CTLR.UE is implemented and was set to 1 when an Uncorrected error was detected.
- ERR<n>CTLR.UE is not implemented and the node always reports errors.

It is IMPLEMENTATION DEFINED whether this bit can be set to 1 by a Deferred error. This bit is not valid and reads UNKNOWN if any of:

- ERR<*n*>STATUS.UE is set to 0 and this bit is only set to 1 by Uncorrected errors.
- ERR<*n*>STATUS.{UE, DE} are both set to 0 and this bit can be set to 1 by Deferred errors.
- ERR<*n*>STATUS.V is set to 0.

This bit is read/write-one-to-clear.

**OF, bit [27]**
Overflow.

Indicates that multiple errors have been detected. This bit is set to 1 when one of the following occurs:

- A corrected error counter is implemented, an error is counted, and the counter overflows.
- A corrected error counter is not implemented, a corrected error is recorded, and ERR<*n*>STATUS.V was previously set to 1.

A type of error other than a corrected error is recorded and ERR<*n*>STATUS.V was previously set to 1. .

Otherwise, this bit is unchanged when an error is recorded.

If a corrected error counter is implemented:

- A direct write that modifies the counter overflow flag indirectly might set this bit to an UNKNOWN value.
- A direct write to this bit that clears this bit to zero might indirectly set the counter overflow flag to an UNKNOWN value.

The possible values of this bit are:

**0b0**
No error syndrome has been discarded and, if a Corrected error counter is implemented, it has not overflowed since this bit was last cleared to zero.

**0b1**
At least one error syndrome has been discarded or, if a Corrected error counter is implemented, it might have overflowed, since this bit was last cleared to zero.

This bit is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

This bit is read/write-one-to-clear.

**MV, bit [26]**
Miscellaneous Registers Valid.

The possible values of this bit are:

**0b0**
ERR<*n*>MISC* not valid.

**0b1**
The IMPLEMENTATION DEFINED contents of the ERR<*n*>MISC* registers contains additional information for an error recorded by this record.

This bit is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

This bit is read/write-one-to-clear.

**CE, bits [25:24]**
Corrected error or errors.

The possible values of this field are:

**0b00**
No errors were corrected.

**0b01**
At least one transient error was corrected.

**0b10**
At least one error was corrected.

**0b11**
At least one persistent error was corrected.

The mechanism by which a node detects whether a correctable error is transient or persistent is IMPLEMENTATION DEFINED. If no such mechanism is implemented then the node sets this field to `0b10` when an error is corrected.

This field is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

**DE, bit [23]**
Deferred error or errors.

The possible values of this bit are:

**0b0**
No errors were deferred.

**0b1**
At least one error was not corrected and deferred.

Support for deferring errors is IMPLEMENTATION DEFINED.

This bit is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

This bit is read/write-one-to-clear.

**PN, bit [22]**
Poison.

The possible values of this bit are:

**0b0**
Uncorrected or deferred error from a corrupted value.

**0b1**
Uncorrected error or Deferred error from a poisoned value. Indicates that an error occurred because of the detection of a poison value rather because of the detection of a corrupted value.

It is IMPLEMENTATION DEFINED whether a node can distinguish a poisoned value from a corrupted value.

This bit is not valid and reads UNKNOWN if any of:

- ERR<*n*>STATUS.V is set to 0.
- ERR<*n*>STATUS.{CE, UE} are both set to 0.

This bit is read/write-one-to-clear.

**Note**

If a node detects a corrupted value and defers the error by producing poison then this bit is set to `0b0` at the producer node. The value `0b1` might only be an indication of a poisoned value. As in some EDC schemes, it is possible to mistake a corrupted value for a poisoned value.

**UET, bits [21:20]**
Uncorrected Error Type. Describes the state of the component after detecting or consuming an Uncorrected error.

The possible values of this field are:

**0b00**
Uncorrected error, Uncontainable error (UC).

**0b01**
Uncorrected error, Unrecoverable error (UEU).

**0b10**
Uncorrected error, Latent or Restartable error (UEO).

**0b11**
Uncorrected error, Signaled or Recoverable error (UER).

This field is not valid and reads UNKNOWN if any of:

- ERR<*n*>STATUS.V is set to 0.
- ERR<*n*>STATUS.UE is set to 0.

This field is read/write-ones-to-clear. Writing a value other than all-zeros or all-ones sets this field to an UNKNOWN value.

Note: Software might use the information in the error record registers to determine what recovery is necessary.

**CI, bit [19]**
Critical error. Indicates whether a critical error condition has been recorded.

The possible values of this bit are:

**0b0**
No critical error condition.

**0b1**
Critical error condition.

This bit is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

This bit is read/write-one-to-clear.

**Bits [18:16]**
Reserved, RES0.

**IERR, bits [15:8]**
IMPLEMENTATION DEFINED error code.

Used with any primary error code SERR value. Further IMPLEMENTATION DEFINED information can be placed in the MISC registers.

The subset of architecturally-defined values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register then the value read back from this field is UNKNOWN.

This field is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

**Note**

One or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

**SERR, bits [7:0]**

Architecturally-defined primary error code. The primary error code might be used by a fault handling agent to triage an error without requiring device-specific code. For example, to count and threshold corrected errors in software, or generate a short log entry.

The possible values of this field are:

**0**

No error.

**1**

IMPLEMENTATION DEFINED error.

**2**

Data value from (non-associative) internal memory. For example, ECC from on-chip SRAM or buffer.

**3**

IMPLEMENTATION DEFINED pin.

**4**

Assertion failure. For example, consistency failure.

**5**

Internal data path. For example, parity on ALU result.

**6**

Data value from associative memory. For example, ECC error on cache data.

**7**

Address/control value or values from associative memory. For example, ECC error on cache tag.

**10**

Data value from producer. For example, parity error on write data bus.

**11**

Address/control value or values from producer. For example, parity error on address bus.

**12**

Data value from (non-associative) external memory. For example, ECC error in SDRAM.

**13**

Illegal address (software fault). For example, access to unpopulated memory.

**14**

Illegal access (software fault). For example, byte write to word register.

**15**

Illegal state (software fault). For example, device not ready.

**16**

Internal data register. For example, parity on a FP&MVE register. For a PE, all general-purpose, stack pointer, and FP&MVE registers are data registers.

**17**

Internal control register. For example, Parity on a system register. For a PE, all registers other than general-purpose, stack pointer, and FP&MVE registers are control registers.

**18**

Error response from slave. For example, error response from cache write-back.

**19**

External timeout. For example, timeout on interaction with another node.

**20**

Internal timeout. For example, timeout on interface within the node.

**21**

Deferred error from slave not supported at master. For example, poisoned data received from a slave by a master that cannot defer the error further.

All other values are reserved.

The subset of architecturally-defined values that this field can take is IMPLEMENTATION DEFINED. If any value not in this set is written to this register then the value read back from this field is UNKNOWN.

This field is not valid and reads UNKNOWN if ERR<*n*>STATUS.V is set to 0.

Note: one or more bits of this field might be implemented as fixed read-as-zero or read-as-one values.

## D1.2.95 EXC_RETURN, Exception Return Payload

The EXC_RETURN characteristics are:

**Purpose**
Value provided in LR on entry to an exception, and used with a BX or load to PC to perform an exception return.

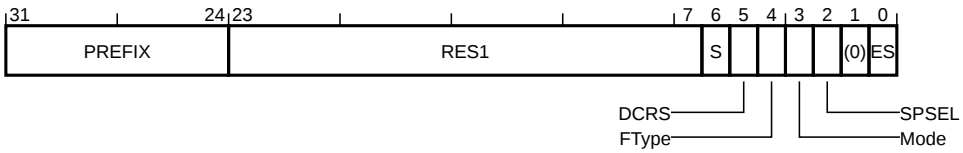
**Usage constraints**
None.

**Configurations**
All.

**Attributes**
32-bit payload.

### Field descriptions

The EXC_RETURN bit assignments are:




**PREFIX, bits [31:24]**
Prefix. Indicates that this is an EXC_RETURN value.

This field reads as `0b11111111`.

**Bits [23:7]**
Reserved, RES1.

**S, bit [6]**
Secure or Non-secure stack. Indicates whether a Secure or Non-secure stack is used to restore stack frame on exception return.

The possible values of this bit are:

**0**
Non-secure stack used.

**1**
Secure stack used.

If the Security Extension is not implemented, this bit is UNPREDICTABLE.

**DCRS, bit [5]**
Default callee register stacking. Indicates whether the default stacking rules apply, or whether the callee registers are already on the stack.

The possible values of this bit are:

**0**
Stacking of the callee saved registers skipped.

**1**
Default rules for stacking the callee registers followed.

**FType, bit [4]**
Stack frame type. Indicates whether the stack frame is a standard integer only stack frame or an extended Floating-point stack frame.

The possible values of this bit are:

**0**
Extended stack frame.

**1**
Standard stack frame.

If the Floating-point Extension is not implemented, this bit is RES1.

**Mode, bit [3]**
Mode. Indicates the Mode that was stacked from.

The possible values of this bit are:

**0**
Handler mode.

**1**
Thread mode.

**SPSEL, bit [2]**
Stack pointer selection. The value of this bit indicates the transitory value of the CONTROL.SPSEL bit associated with the Security state of the exception as indicated by EXC_RETURN.ES.

The possible values of this bit are:

**0**
Main stack pointer.

**1**
Process stack pointer.

**Bit [1]**
Reserved, RES0.

**ES, bit [0]**
Exception Secure. The security domain the exception was taken to.

The possible values of this bit are:

**0**
Non-secure.

**1**
Secure.

If the Security Extension is not implemented, this bit is UNPREDICTABLE.

## D1.2.96 FAULTMASK, Fault Mask Register

The FAULTMASK characteristics are:

**Purpose**
Provides access to the PE FAULTMASK register.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**
Present only if the Main Extension is implemented.

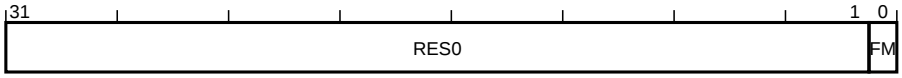
This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states.

### Field descriptions

The FAULTMASK bit assignments are:




**Bits [31:1]**
Reserved, RES0.

**FM, bit [0]**
Fault mask enable. The Secure and Non-secure FAULTMASK registers individually boost the current execution priority based on the settings of AIRCR.PRIS and AIRCR.BFHFNMINS. If AIRCR.BFHFNMINS is zero, AIRCR.PRIS is zero, and FAULTMASK_NS.FM is one, the execution priority is boosted to 0. If AIRCR.BFHFNMINS is zero, AIRCR.PRIS is one, and FAULTMASK_NS.FM is one, the execution priority is boosted to `0x80`. If AIRCR.BFHFNMINS is zero and FAULTMASK_S is one, the execution priority is boosted to -1. If AIRCR.BFHFNMINS is one and FAULTMASK_NS is one, the execution priority is boosted to -1. If AIRCR.BFHFNMINS is one and FAULTMASK_S is one, the execution priority is boosted to -3.

The possible values of this bit are:

**0**
No effect.

**1**
Boost priority.

On an exception return from a raw execution priority greater or equal to zero, the FM bit corresponding to EXC_RETURN.ES is cleared.

This bit resets to zero on a Warm reset.

## D1.2.97 FNC_RETURN, Function Return Payload

The FNC_RETURN characteristics are:

**Purpose**
Value provided in LR on entry to Non-secure state from a Secure BLXNS.

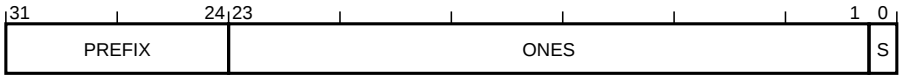
**Usage constraints**
None.

**Configurations**
All.

**Attributes**
32-bit payload.

### Field descriptions

The FNC_RETURN bit assignments are:




**PREFIX, bits [31:24]**
This field reads as `0b11111110`.

**ONES, bits [23:1]**
This field reads as `0b11111111111111111111111`.

**S, bit [0]**
Secure. Indicates whether the function call was from the Non-secure or Secure state. Because FNC_RETURN is only used when calling from the Secure state, this bit is always set to 1. However, some function chaining cases can result in an SG instruction clearing this bit, so the architecture ignores the state of this bit when processing a branch to FNC_RETURN.

The possible values of this bit are:

**0**
From Non-secure state.

**1**
From Secure state.

## D1.2.98 FPCAR, Floating-Point Context Address Register

The FPCAR characteristics are:

**Purpose**
Holds the location of the unpopulated Floating-point register space allocated on an exception stack frame.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

**Attributes**
32-bit read/write register located at `0xE000EF38`.

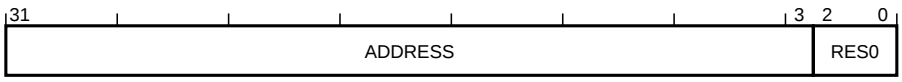
Secure software can access the Non-secure version of this register via FPCAR_NS located at `0xE002EF38`. The location `0xE002EF38` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The FPCAR bit assignments are:




**ADDRESS, bits [31:3]**
Address. The location of the unpopulated Floating-point register space allocated on an exception stack frame.

This field resets to an UNKNOWN value on a Warm reset.

**Bits [2:0]**
Reserved, RES0.

## D1.2.99 FPCCR, Floating-Point Context Control Register

The FPCCR characteristics are:

**Purpose**
Holds control data for the Floating Point Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

**Attributes**
32-bit read/write register located at `0xE000EF34`.

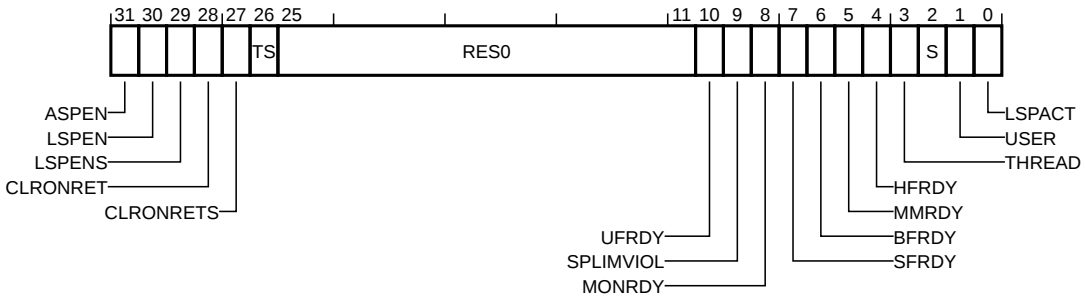
Secure software can access the Non-secure version of this register via FPCCR_NS located at `0xE002EF34`. The location `0xE002EF34` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The FPCCR bit assignments are:




**ASPEN, bit [31]**
Automatic state preservation enable. When this bit is set to 1, execution of a Floating-point instruction sets the CONTROL.FPCA bit to 1.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Executing an FP instruction has no effect on CONTROL.FPCA.

**1**
Executing an FP instruction sets CONTROL.FPCA to 1.

Setting this bit to 1 means the hardware automatically preserves Floating-point context on exception entry and restores it on exception return. As of version ARMv8.1-M of the architecture ARM deprecates setting this field to 0.

This bit resets to one on a Warm reset.

**LSPEN, bit [30]**
Lazy state preservation enable. Enables lazy context save of Floating-point state.

The possible values of this bit are:

**0**
Disable automatic lazy context save.

**1**
Enable automatic lazy context save.

Writes to this bit from Non-secure state are ignored if LSPENS is set to 1.

This bit resets to one on a Warm reset.

**LSPENS, bit [29]**
Lazy state preservation enable Secure. This bit controls whether the LSPEN bit is writable from the Non-secure state. This behaves as RAZ/WI when accessed from the Non-secure state.

The possible values of this bit are:

**0**
LSPEN is readable and writable from both Security states.

**1**
LSPEN is readable from both Security states, but writes to LSPEN are ignored from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**CLRONRET, bit [28]**
Clear on return. Clear Floating-point caller saved registers on exception return.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

When set to 1 the caller saved Floating-point registers (S0 to S15, FPSCR, and VPR) are cleared on exception return (including tail chaining) if CONTROL.FPCA is set to 1 and FPCCR_S.LSPACT is set to 0. Writes to this bit from Non-secure state are ignored if CLRONRETS is set to one.

This bit resets to zero on a Warm reset.

**CLRONRETS, bit [27]**
Clear on return, Secure only. This bit controls whether the CLRONRET bit is writable from the Non-secure state.

The possible values of this bit are:

**0**
The CLRONRET field is accessibly from both Security states.

**1**
The Non-secure view of the CLRONRET field is read-only.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**TS, bit [26]**
Treat as Secure. Treat Floating-point registers as Secure enable.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

When set to 0 the Floating-point registers are treated as Non-secure even when the PE is in Secure state and, therefore, the callee saved registers are never pushed to the stack. If the Floating-point registers never contain data that needs to be protected, clearing this flag can reduce interrupt latency. As this field changes how secure stack frames are interpreted, UNPREDICTABLE behavior can result if the state of this bit is not consistent with the current Secure stacks. Therefore, firmware must take care when modifying this value. This field behaves as RAZ/WI from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**Bits [25:11]**
Reserved, RES0.

**UFRDY, bit [10]**
UsageFault enable. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the UsageFault exception to pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Not able to set the UsageFault exception to pending.

**1**
Able to set the UsageFault exception to pending.

This bit resets to an UNKNOWN value on a Warm reset.

**SPLIMVIOL, bit [9]**
Stack pointer limit violation. This bit indicates whether the Floating-point context violates the stack pointer limit that was active when lazy state preservation was activated. SPLIMVIOL modifies the lazy Floating-point state preservation behavior.

This bit is banked between Security states.

The possible values of this bit are:

**0**
The existing behavior is retained.

**1**
The memory accesses associated with the Floating-point state preservation are not performed. However if the Floating-point state is Secure and FPCCR.TS is set to 1 the registers are still zeroed and the Floating-point state is lost.

This bit resets to an UNKNOWN value on a Warm reset.

**MONRDY, bit [8]**
DebugMonitor ready. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the DebugMonitor exception to pending.

The possible values of this bit are:

**0**
Not able to set the DebugMonitor exception to pending.

**1**
Able to set the DebugMonitor exception to pending.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

This bit resets to an UNKNOWN value on a Warm reset.

**SFRDY, bit [7]**
SecureFault ready. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the SecureFault exception to pending.

This bit is RAZ/WI from Non-secure state.

This bit resets to an UNKNOWN value on a Warm reset.

**BFRDY, bit [6]**
BusFault ready. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the BusFault exception to pending.

The possible values of this bit are:

**0**
Not able to set the BusFault exception to pending.

**1**
Able to set the BusFault exception to pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to an UNKNOWN value on a Warm reset.

**MMRDY, bit [5]**
MemManage ready. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the MemManage exception to pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Not able to set the MemManage exception to pending.

**1**
Able to set the MemManage exception to pending.

This bit resets to an UNKNOWN value on a Warm reset.

**HFRDY, bit [4]**
HardFault ready. Indicates whether the software executing, when the PE allocated the Floating-point stack frame, was able to set the HardFault exception to pending.

The possible values of this bit are:

**0**
Not able to set the HardFault exception to pending.

**1**
Able to set the HardFault exception to pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to an UNKNOWN value on a Warm reset.

**THREAD, bit [3]**
Thread mode. Indicates the PE mode when it allocated the Floating-point stack frame.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Handler mode.

**1**
Thread mode.

This bit is for fault handler information only and does not interact with the exception model.

This bit resets to an UNKNOWN value on a Warm reset.

**S, bit [2]**
Security. Security status of the Floating-point context. This bit is only present in the Secure version of the register. This bit is updated whenever lazy state preservation is activated, or when a Floating-point instruction is executed.

The possible values of this bit are:

**0**
Indicates the Floating-point context belongs to the Non-secure state.

**1**
Indicates the Floating-point context belongs to the Secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to one on a Warm reset.

**USER, bit [1]**
User privilege. Indicates the privilege level of the software executing when the PE allocated the Floating-point stack frame.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Privileged.

**1**
Unprivileged.

This bit resets to an UNKNOWN value on a Warm reset.

**LSPACT, bit [0]**
Lazy state preservation active. Indicates whether lazy preservation of the Floating-point state is active.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Lazy state preservation is not active.

**1**
Lazy state preservation is active.

This bit resets to zero on a Warm reset.

## D1.2.100 FPCXT, Floating-point context payload

The FPCXT characteristics are:

**Purpose**
Values produced or consumed by instructions that provide access to the Floating-point context.

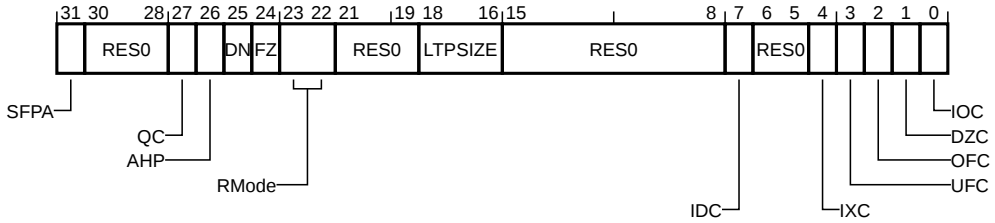
**Usage constraints**
None.

**Configurations**
All.

**Attributes**
32-bit payload.

### Field descriptions

The FPCXT bit assignments are:




**SFPA, bit [31]**
Secure Floating-point active. The value corresponds to CONTROL.SFPA.

**Bits [30:28]**
Reserved, RES0.

**QC, bit [27]**
Cumulative saturation bit. The value corresponds to FPSCR.QC.

**AHP, bit [26]**
Alternative half-precision control bit. The value corresponds to FPSCR.AHP.

**DN, bit [25]**
Default NaN mode control bit. The value corresponds to FPSCR.DN.

**FZ, bit [24]**
Flush-to-zero mode control for single and double precision Floating-point. The value corresponds to FPSCR.FZ.

**RMode, bits [23:22]**
Rounding mode control field. The value corresponds to FPSCR.RMode.

**Bits [21:19]**
Reserved, RES0.

**LTPSIZE, bits [18:16]**
The vector element size that is used when applying low-overhead-loop tail predication to vector instructions.

**Bits [15:8]**
Reserved, RES0.

**IDC, bit [7]**
Input Denormal cumulative exception bit. The value corresponds to FPSCR.IDC.

**Bits [6:5]**
Reserved, RES0.

**IXC, bit [4]**
Inexact cumulative exception bit. The value corresponds to FPSCR.IXC.

**UFC, bit [3]**
Underflow cumulative exception bit. The value corresponds to FPSCR.UFC.

**OFC, bit [2]**
Overflow cumulative exception bit. The value corresponds to FPSCR.OFC.

**DZC, bit [1]**
Divide by Zero cumulative exception bit. The value corresponds to FPSCR.DZC.

**IOC, bit [0]**
Invalid Operation cumulative exception bit. The value corresponds to FPSCR.IOC.

## D1.2.101 FPDSCR, Floating-Point Default Status Control Register

The FPDSCR characteristics are:

**Purpose**

Holds the default values for the Floating-point status control data that the PE assigns to FPSCR when it creates a new Floating-point context.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

**Attributes**

32-bit read/write register located at `0xE000EF3C`.

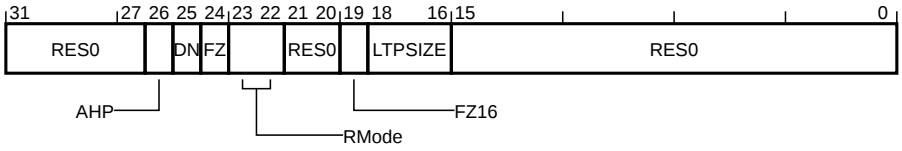
Secure software can access the Non-secure version of this register via FPDSCR_NS located at `0xE002EF3C`. The location `0xE002EF3C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The FPDSCR bit assignments are:




**Bits [31:27]**

Reserved, RES0.

**AHP, bit [26]**

Alternative half-precision. Default value for FPSCR.AHP.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**DN, bit [25]**

Default NaN. Default value for FPSCR.DN.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**FZ, bit [24]**

Flush-to-zero. Default value for FPSCR.FZ.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**RMode, bits [23:22]**
Rounding mode. Default value for FPSCR.RMode.

If the Floating-point Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

**Bits [21:20]**
Reserved, RES0.

**FZ16, bit [19]**
Flush-to-zero mode control bit on half-precision data-processing instructions. Default value for FPSCR.FZ16.

This bit resets to zero on a Warm reset.

**LTPSIZE, bits [18:16]**
The vector element size used when applying low-overhead-loop tail predication to vector instructions. Default value for FPSCR.LTPSIZE.

If the Low Overhead Branch Extension is not implemented, this field is RES0.

This field reads as `0x4`.

**Bits [15:0]**
Reserved, RES0.

## D1.2.102 FPSCR, Floating-point Status and Control Register

The FPSCR characteristics are:

**Purpose**
Provides control of the Floating-point unit.

**Usage constraints**
Privileged and unprivileged access permitted.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

**Attributes**
32-bit read/write special-purpose register.

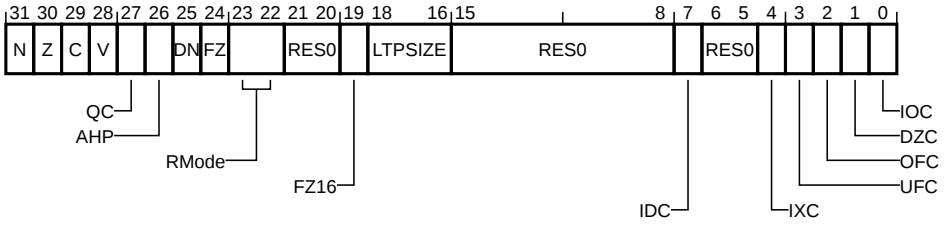
This register is not banked between Security states.

### Preface

Writes to FPSCR can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to FPSCR write. This means that they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

### Field descriptions

The FPSCR bit assignments are:




**N, bit [31]**
Negative condition flag. When updated by a VCMP instruction, this bit indicates whether the result was less than.

The possible values of this bit are:

**0**
Compare result was not less than.

**1**
Compare result was less than.

See VCMP for details.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**Z, bit [30]**
Zero condition flag. When updated by a VCMP instruction, this bit indicates whether the result was equal.

The possible values of this bit are:

**0**
Compare result was not equal.

**1**
Compare result was equal.

See VCMP for details.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**C, bit [29]**
Carry condition flag. Accessed by the VCMP, VADC, and VSBC instructions. For VCMP this bit indicates whether the result was not less than. In VADC and VSBC this bit is used to hold the carry in/out flag.

The possible values of this bit are:

**0**
Compare result was less than.

**1**
Compare result was not less than.

See VCMP, VADC, and VSBC for details.

This bit resets to an UNKNOWN value on a Warm reset.

**V, bit [28]**
Overflow condition flag. When updated by a VCMP instruction, this bit indicates whether the result was unordered.

The possible values of this bit are:

**0**
Compare result was not unordered.

**1**
Compare result was unordered.

See VCMP for details.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**QC, bit [27]**
Cumulative saturation bit. This bit is set to 1 to indicate that an MVE integer operation has saturated since 0 was last written to this bit.

If MVE is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**AHP, bit [26]**
Alternative half-precision control bit. This bit controls how the PE interprets 16-bit Floating-point values.

The possible values of this bit are:

**0**
IEEE half-precision format selected.

**1**
Alternative half-precision format selected.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**DN, bit [25]**
Default NaN mode control bit. This bit determines whether Floating-point operations propagate NaNs or use the Default NaN.

The possible values of this bit are:

**0**
NaN operands propagate through to the output of a Floating-point operation.

**1**
Any operation involving one of more NaNs returns the Default NaN.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**FZ, bit [24]**
Flush-to-zero mode control for single and double precision Floating-point. This bit determines whether denormal Floating-point values are treated as though they are zero.

The possible values of this bit are:

**0**
Flush-to-zero mode disabled. Behavior of the Floating-point unit is fully compliant with the IEEE754 standard.

**1**
Flush-to-zero mode enabled.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

**RMode, bits [23:22]**
Rounding mode control field. This field determines what rounding mode is applied to Floating-point operations.

The possible values of this field are:

**0b00**
Round to Nearest (RN) mode.

**0b01**
Round towards Plus Infinity (RP) mode.

**0b10**
Round towards Minus Infinity (RM) mode.

**0b11**
Round towards Zero (RZ) mode.

If the Floating-point Extension is not implemented, this field is RAZ/WI.

This field resets to an UNKNOWN value on a Warm reset.

**Bits [21:20]**
Reserved, RES0.

**FZ16, bit [19]**
Flush-to-zero mode control bit on half-precision data-processing instructions.

The possible values of this bit are:

**0**
Flush-to-zero mode disabled. Behavior of the Floating-point unit is fully compliant with the IEEE 754 standard.

**1**
Flush-to-zero mode enabled.

The value of this bit applies to both scalar and MVE Floating-point half-precision calculations.

This bit resets to an UNKNOWN value on a Warm reset.

**LTPSIZE, bits [18:16]**
The vector element size used when applying low-overhead-loop tail predication to vector instructions.

The possible values of this field are:

**0b000**
8 bits.

**0b001**
16 bits.

**0b010**
32 bits.

**0b011**
64 bits.

**0b100**
Tail predication not applied.

All other values are reserved.

The loop hardware behaves as if this field had the value 4 (indicating no low-overhead-loop predication) if no FP context is active. This field reads as 4 and ignores writes if MVE is not implemented.

If the Low Overhead Branch Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

**Bits [15:8]**
Reserved, RES0.

**IDC, bit [7]**
Input Denormal cumulative exception bit. This sticky flag records whether a Floating-point input denormal exception has been detected since last cleared.

The possible values of this bit are:

**0**
Input Denormal exception has not occurred since 0 was last written to this bit.

**1**
Input Denormal exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [6:5]**
Reserved, RES0.

**IXC, bit [4]**
Inexact cumulative exception bit. This sticky flag records whether a Floating-point inexact exception has been detected since last cleared.

The possible values of this bit are:

**0**
Inexact exception has not occurred since 0 was last written to this bit.

**1**
Inexact exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**UFC, bit [3]**
Underflow cumulative exception bit. This sticky flag records whether a Floating-point Underflow exception has been detected since last cleared.

The possible values of this bit are:

**0**
Underflow exception has not occurred since 0 was last written to this bit.

**1**
Underflow exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

**OFC, bit [2]**
Overflow cumulative exception bit. This sticky flag records whether a Floating-point overflow exception has been detected since last cleared.

The possible values of this bit are:

**0**
Overflow exception has not occurred since 0 was last written to this bit.

**1**
Overflow exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**DZC, bit [1]**
Divide by Zero cumulative exception bit. This sticky flag records whether a Floating-point divide by zero exception has been detected since last cleared.

The possible values of this bit are:

**0**
Division by Zero exception has not occurred since 0 was last written to this bit.

**1**
Division by Zero exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**IOC, bit [0]**
Invalid Operation cumulative exception bit. This sticky flag records whether a Floating-point invalid operation exception has been detected since last cleared.

The possible values of this bit are:

**0**
Invalid Operation exception has not occurred since 0 was last written to this bit.

**1**
Invalid Operation exception has occurred since 0 was last written to this bit.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

### D1.2.103 FP_CIDR0, FP Component Identification Register 0

The FP_CIDR0 characteristics are:

**Purpose**

Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

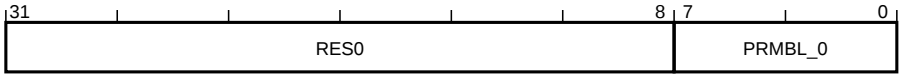
**Attributes**

32-bit read-only register located at `0xE0002FF0`.

This register is not banked between Security states.

## Field descriptions

The FP_CIDR0 bit assignments are:




**Bits [31:8]**

Reserved, RES0.

**PRMBL_0, bits [7:0]**

CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

### D1.2.104 FP_CIDR1, FP Component Identification Register 1

The FP_CIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

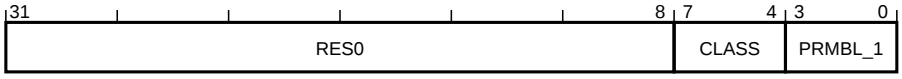
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FF4`.

This register is not banked between Security states.

## Field descriptions

The FP_CIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

## D1.2.105 FP_CIDR2, FP Component Identification Register 2

The FP_CIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

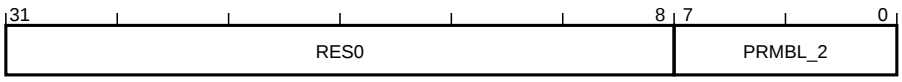
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FF8`.

This register is not banked between Security states.

### Field descriptions

The FP_CIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

### D1.2.106 FP_CIDR3, FP Component Identification Register 3

The FP_CIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

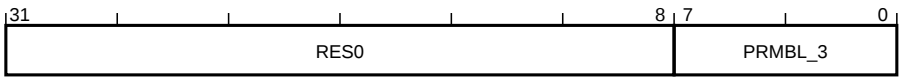
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FFC`.

This register is not banked between Security states.

## Field descriptions

The FP_CIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

### D1.2.107 FP_COMPn, Flash Patch Comparator Register, n = 0 - 125

The FP_COMP{0..125} characteristics are:

**Purpose**
Holds an address for comparison.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the FPB is implemented.

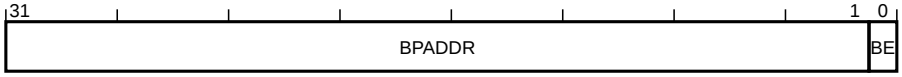
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read/write register located at `0xE0002008` + 4*n*.

This register is not banked between Security states.

## Field descriptions

The FP_COMP{0..125} bit assignments are:




**BPADDR, bits [31:1]**
Breakpoint address. Specifies bits[31:1] of the breakpoint instruction address.

**BE, bit [0]**
Breakpoint enable. Selects between remapping and breakpoint functionality.

The possible values of this bit are:

**0**
Breakpoint disabled.

**1**
Breakpoint enabled.

For backwards compatibility, when disabling a breakpoint software must write zero to the whole register.

This bit resets to zero on a Cold reset.

## D1.2.108 FP_CTRL, Flash Patch Control Register

The FP_CTRL characteristics are:

**Purpose**
Provides FPB implementation information, and the global enable for the FPB unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the FPB is implemented.

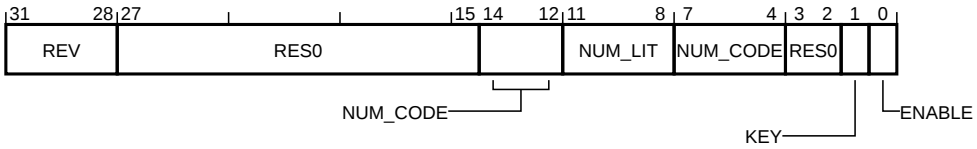
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read/write register located at `0xE0002000`.

This register is not banked between Security states.

### Field descriptions

The FP_CTRL bit assignments are:




**REV, bits [31:28]**
Revision. Flash Patch and Breakpoint Unit architecture revision.

The possible values of this field are:

**`0b0001`**
Flash Patch Breakpoint version 2 implemented.

All other values are reserved.

This field is read-only.

This field reads as `0b0001`.

**Bits [27:15]**
Reserved, RES0.

**NUM_CODE, bits [14:12,7:4]**
Number of implemented code comparators. Indicates the number of implemented instruction address comparators. Zero indicates no Instruction Address comparators are implemented. The Instruction Address comparators are numbered from 0 to NUM_CODE - 1.

This field is read-only.

This field reads as an IMPLEMENTATION DEFINED value.

**NUM_LIT, bits [11:8]**
Number of literal comparators. This field is RAZ/WI. Remapping is not supported in Armv8-M.

**Bits [3:2]**
Reserved, RES0.

**KEY, bit [1]**
FP_CTRL write-enable key. Writes to the FP_CTRL are ignored unless KEY is concurrently written to one.

The possible values of this bit are:

**0**
Concurrent write to FP_CTRL ignored.

**1**
Concurrent write to FP_CTRL permitted.

This bit reads-as-zero.

**ENABLE, bit [0]**
Flash Patch global enable. Enables the FPB.

The possible values of this bit are:

**0**
All FPB functionality disabled.

**1**
FPB enabled.

This bit resets to zero on a Cold reset.

### D1.2.109 FP_DEVARCH, FPB Device Architecture Register

The FP_DEVARCH characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

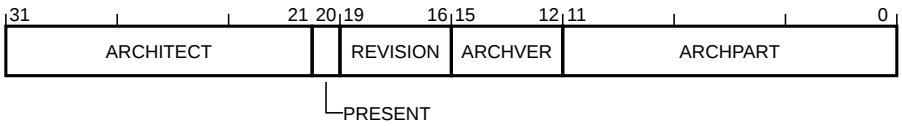
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FBC`.

This register is not banked between Security states.

## Field descriptions

The FP_DEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**
Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

**`0x23B`**
JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

**PRESENT, bit [20]**
DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

**1**
DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**
Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**
FPB architecture v2.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**
Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**
FPB architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0001.

**ARCHPART, bits [11:0]**
Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA03**
FPB architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA03.

### D1.2.110 FP_DEVTYPE, FPB Device Type Register

The FP_DEVTYPE characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

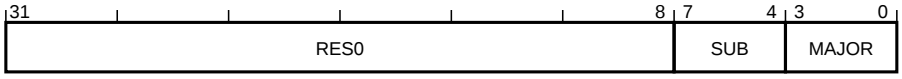
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FCC`.

This register is not banked between Security states.

## Field descriptions

The FP_DEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Sub-type. Component sub-type.

The possible values of this field are:

**`0x0`**
Other.

This field reads as `0b0000`.

**MAJOR, bits [3:0]**
Major type. Component major type.

The possible values of this field are:

**`0x0`**
Miscellaneous.

This field reads as `0b0000`.

### D1.2.111 FP_LAR, FPB Software Lock Access Register

The FP_LAR characteristics are:

**Purpose**

Provides CoreSight Software Lock control for the FPB, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

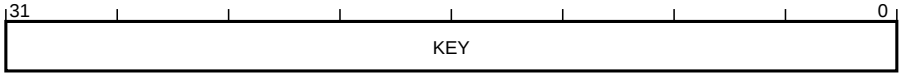
**Attributes**

32-bit write-only register located at `0xE0002FB0`.

This register is not banked between Security states.

## Field descriptions

The FP_LAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.112 FP_LSR, FPB Software Lock Status Register

The FP_LSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the FPB, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

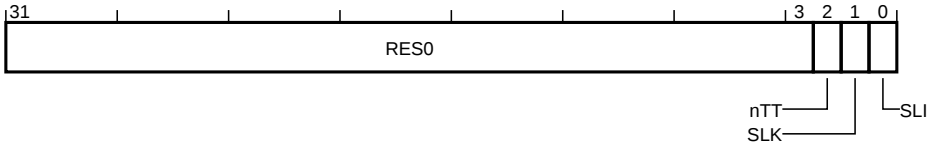
**Attributes**

32-bit read-only register located at `0xE0002FB4`.

This register is not banked between Security states.

## Field descriptions

The FP_LSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

### D1.2.113 FP_PIDR0, FP Peripheral Identification Register 0

The FP_PIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

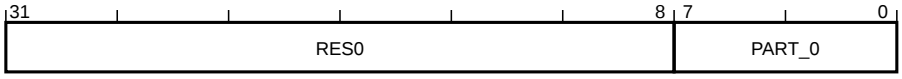
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FE0`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.114 FP_PIDR1, FP Peripheral Identification Register 1

The FP_PIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

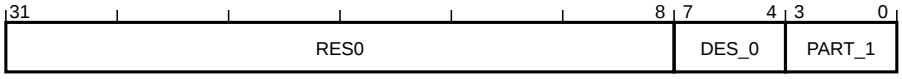
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FE4`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.115 FP_PIDR2, FP Peripheral Identification Register 2

The FP_PIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

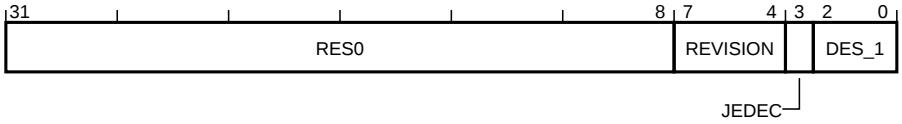
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FE8`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

**DES_1, bits [2:0]**
JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.116 FP_PIDR3, FP Peripheral Identification Register 3

The FP_PIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

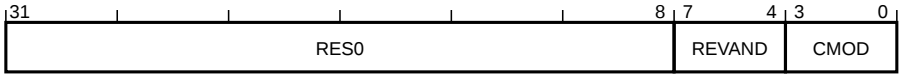
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FEC`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.117 FP_PIDR4, FP Peripheral Identification Register 4

The FP_PIDR4 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

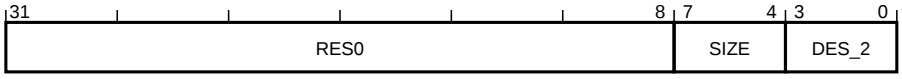
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FD0`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

**DES_2, bits [3:0]**
JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.118 FP_PIDR5, FP Peripheral Identification Register 5

The FP_PIDR5 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

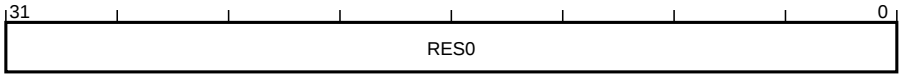
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FD4`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.119 FP_PIDR6, FP Peripheral Identification Register 6

The FP_PIDR6 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

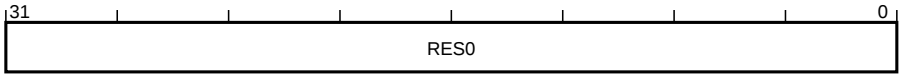
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FD8`.

This register is not banked between Security states.

## Field descriptions

The FP_PIDR6 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

## D1.2.120 FP_PIDR7, FP Peripheral Identification Register 7

The FP_PIDR7 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the Flash Patch and Breakpoint Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

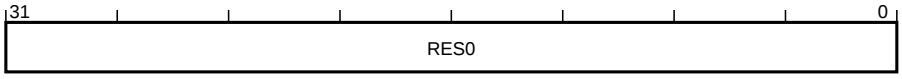
This register is RES0 if the FPB is not implemented.

**Attributes**
32-bit read-only register located at `0xE0002FDC`.

This register is not banked between Security states.

### Field descriptions

The FP_PIDR7 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.121 FP_REMAP, Flash Patch Remap Register

The FP_REMAP characteristics are:

**Purpose**

Indicates whether the implementation supports Flash Patch remap and, if it does, holds the target address for remap.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

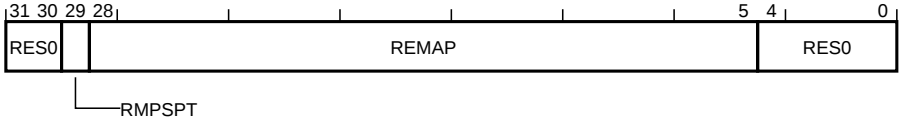
**Attributes**

32-bit read-only register located at `0xE0002004`.

This register is not banked between Security states.

## Field descriptions

The FP_REMAP bit assignments are:




**Bits [31:30]**

Reserved, RES0.

**RMPSPT, bit [29]**

Remap supported. This field is RAZ. Remapping is not supported in Armv8-M.

**REMAP, bits [28:5]**

Remap address.

Reserved, RES0.

**Bits [4:0]**

Reserved, RES0.

### D1.2.122 HFSR, HardFault Status Register

The HFSR characteristics are:

**Purpose**
Shows the cause of any HardFaults.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write-one-to-clear register located at `0xE000ED2C`.

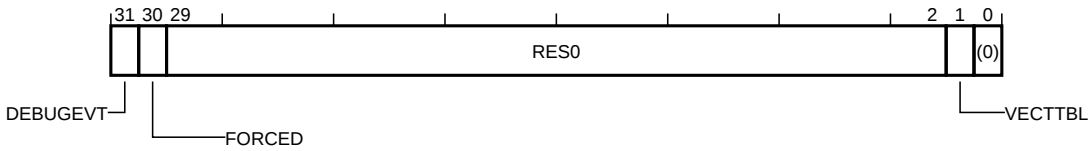
Secure software can access the Non-secure version of this register via HFSR_NS located at `0xE002ED2C`. The location `0xE002ED2C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The HFSR bit assignments are:




**DEBUGEVT, bit [31]**
Debug event. Indicates when a debug event has occurred.

The possible values of this bit are:

**0**
No debug event has occurred.

**1**
Debug event has occurred. The Debug Fault Status Register has been updated.

The PE sets this bit to 1 only when Halting debug is disabled and a debug event occurs. When AIRCR.BFHFNMINS is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**FORCED, bit [30]**
Forced. Indicates that a fault with configurable priority has been escalated to a HardFault exception, because it could not be made active, because of priority, or because it was disabled.

The possible values of this bit are:

**0**
No priority escalation has occurred.

**1**
Processor has escalated a configurable-priority exception to HardFault.

When AIRCR.BFHFNMINS is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**Bits [29:2]**
Reserved, RES0.

**VECTTBL, bit [1]**
Vector table. Indicates when a fault has occurred because of a vector table read error on exception processing.

The possible values of this bit are:

**0**
No vector table read fault has occurred.

**1**
Vector table read fault has occurred.

When AIRCR.BFHFNMINS is set to zero, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

**Bit [0]**
Reserved, RES0.

### D1.2.123 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

**Purpose**
Invalidate all instruction caches to PoU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

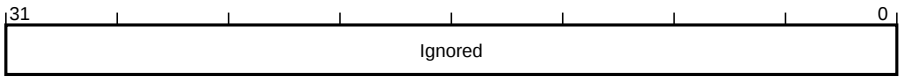
**Attributes**
32-bit write-only register located at `0xE000EF50`.

Secure software can access the Non-secure version of this register via ICIALLU_NS located at `0xE002EF50`. The location `0xE002EF50` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ICIALLU bit assignments are:




**Ignored, bits [31:0]**
The value written to this field is ignored. Ignored.

### D1.2.124 ICIMVAU, Instruction Cache line Invalidate by Address to PoU

The ICIMVAU characteristics are:

**Purpose**
Invalidate instruction cache line by address to PoU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
This register is always implemented.

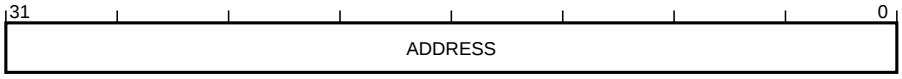
**Attributes**
32-bit write-only register located at `0xE000EF58`.

Secure software can access the Non-secure version of this register via ICIMVAU_NS located at `0xE002EF58`. The location `0xE002EF58` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ICIMVAU bit assignments are:




**ADDRESS, bits [31:0]**
Address. Writing to this field initiates the maintenance operation for the address written.

## D1.2.125 ICSR, Interrupt Control and State Register

The ICSR characteristics are:

**Purpose**
Controls and provides status information for NMI, PendSV, SysTick and interrupts.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

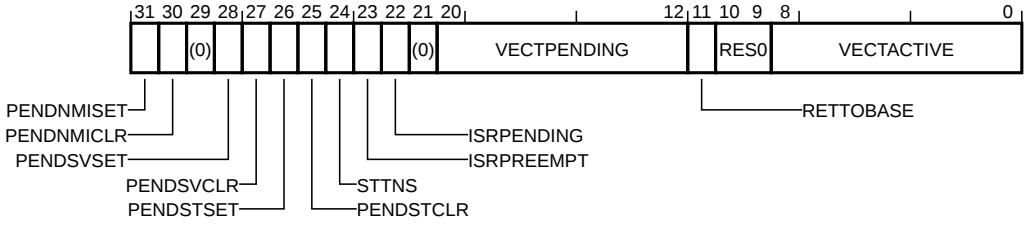
**Attributes**
32-bit read/write register located at `0xE000ED04`.

Secure software can access the Non-secure version of this register via ICSR_NS located at `0xE002ED04`. The location `0xE002ED04` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

### Field descriptions

The ICSR bit assignments are:




**PENDNMISET, bit [31], on a write**
Pend NMI set. Allows the NMI exception to be set as pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Sets the NMI exception pending.

If both PENDNMISET and PENDNMICLR are written to one simultaneously, the pending state of the NMI exception becomes UNKNOWN.

This bit is write-one-to-set. Writes of zero are ignored.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

**PENDNMISET, bit [31], on a read**
Pend NMI set. Indicates whether the NMI exception is pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
NMI exception not pending.

**1**
NMI exception pending.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**PENDNMICLR, bit [30]**
Pend NMI clear. Allows the NMI exception pending state to be cleared.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

**Bit [29]**
Reserved, RES0.

**PENDSVSET, bit [28], on a write**
Pend PendSV set. Allows the PendSV exception for the selected Security state to be set as pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Sets the PendSV exception pending.

If both PENDSVSET and PENDSVCLR are written to one simultaneously, the pending state of the associated PendSV exception becomes UNKNOWN.

This bit is write-one-to-set. Writes of zero are ignored.

**PENDSVSET, bit [28], on a read**
Pend PendSV set. Indicates whether the PendSV for the selected Security state exception is pending.

This bit is banked between Security states.

The possible values of this bit are:

**0**
PendSV exception not pending.

**1**
PendSV exception pending.

This bit resets to zero on a Warm reset.

**PENDSVCLR, bit [27]**
Pend PendSV clear. Allows the PendSV exception pending state to be cleared for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

**PENDSTSET, bit [26], on a write**
Pend SysTick set. Allows the SysTick for the selected Security state exception to be set as pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Sets the SysTick exception for the selected Security state pending.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

**PENDSTSET, bit [26], on a read**
Pend SysTick set. Indicates whether the SysTick for the selected Security state exception is pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SysTick exception not pending.

**1**
SysTick exception pending.

If both PENDSTSET and PENDSTCLR are written to one simultaneously, the pending state of the associated SysTick exception becomes UNKNOWN.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

**PENDSTCLR, bit [25]**
Pend SysTick clear. Allows the SysTick exception pending state to be cleared for the selected Security state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No effect.

**1**
Clear pending status.

This bit is write-only, and reads-as-zero.

This bit is write-one-to-clear. Writes of zero are ignored.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

**STTNS, bit [24]**
SysTick Targets Non-secure. Controls whether in a single SysTick implementation, the SysTick is Secure or Non-secure.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SysTick is Secure.

**1**
SysTick is Non-secure.

Behaves as RAZ/WI when either no SysTick or both SysTick timers are implemented. In a PE with the Main Extension and Security Extension this bit is RES0. This bit is RAZ/WI when accessed from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**ISRPREEMPT, bit [23]**
Interrupt preempt. Indicates whether a pending exception will be handled on exit from Debug state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Will not handle.

**1**
Will handle a pending exception.

The value of this bit is UNKNOWN when not in Debug state.

This bit is read-only.

If neither Halting debug or the Main Extension are implemented, this bit is RES0.

**ISRPENDING, bit [22]**
Interrupt pending. Indicates whether an external interrupt, generated by the NVIC, is pending.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
No external interrupt pending.

**1**
External interrupt pending.

This bit is read-only.

If neither Halting debug or the Main Extension are implemented, this bit is RES0.

**Note**

The value of DHCSR.C_MASKINTS is ignored.

**Bit [21]**
Reserved, RES0.

**VECTPENDING, bits [20:12]**
Vector pending. The exception number of the highest priority pending and enabled interrupt.

This field is not banked between Security states.

The possible values of this field are:

**Zero**
No pending and enabled exception.

**Non zero**
Exception number.

This value is 1 when read from a Non-secure state and a secure exception is the highest priority pending exception.

This field is read-only.

**Note**

If DHCSR.C_MASKINTS is set, the PendSV, SysTick, and configurable external interrupts are masked and will not be shown as pending in VECTPENDING.

**RETTOBASE, bit [11]**
Return to base. In Handler mode, indicates whether there is more than one active exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
There is more than one active exception.

**1**
There is only one active exception.

In Thread mode the value of this bit is UNKNOWN.

This bit is read-only.

If the Main Extension is not implemented, this bit is RES0.

**Bits [10:9]**
Reserved, RES0.

**VECTACTIVE, bits [8:0]**
Vector active. The exception number of the current executing exception.

This field is not banked between Security states.

The possible values of this field are:

**Zero**
Thread mode.

**Non zero**
Exception number.

This value is the same as the IPSR Exception number. When the IPSR value has been set to 1 because of a function call to Non-secure state, this field is also set to 1.

This field is read-only.

If neither Halting debug or the Main Extension are implemented, this field is RES0.

## D1.2.126 ICTR, Interrupt Controller Type Register

The ICTR characteristics are:

**Purpose**
Provides information about the interrupt controller.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

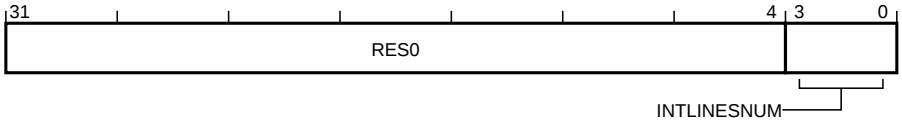
**Attributes**
32-bit read-only register located at `0xE000E004`.

Secure software can access the Non-secure version of this register via ICTR_NS located at `0xE002E004`. The location `0xE002E004` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ICTR bit assignments are:




**Bits [31:4]**
Reserved, RES0.

**INTLINESNUM, bits [3:0]**
Interrupt line set number. Indicates the number of the highest implemented register in each of the NVIC control register sets, or in the case of NVIC_IPR*n*, 4xINTLINESNUM.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.127 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

**Purpose**
Provides information about the IMPLEMENTATION DEFINED features of the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

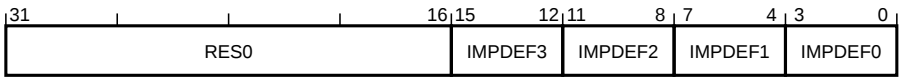
**Attributes**
32-bit read-only register located at `0xE000ED4C`.

Secure software can access the Non-secure version of this register via ID_AFR0_NS located at `0xE002ED4C`. The location `0xE002ED4C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_AFR0 bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**IMPDEF*m*, bits [4*m*+3:4*m*], for *m* = 0 to 3**
IMPLEMENTATION DEFINED. IMPLEMENTATION DEFINED meaning.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.128 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

**Purpose**
Provides top level information about the debug system.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

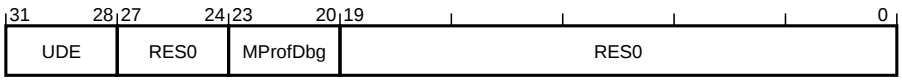
**Attributes**
32-bit read-only register located at `0xE000ED48`.

Secure software can access the Non-secure version of this register via ID_DFR0_NS located at `0xE002ED48`. The location `0xE002ED48` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_DFR0 bit assignments are:




**UDE, bits [31:28]**
Unprivileged Debug Extension. Indicates support for the Unprivileged Debug Extension.

The possible values of this field are:

**`0b0000`**
Unprivileged Debug Extension is not implemented.

**`0b0001`**
Unprivileged Debug Extension is implemented.

**Bits [27:24]**
Reserved, RES0.

**MProfDbg, bits [23:20]**
M-profile debug. Indicates the supported M-profile debug architecture.

The possible values of this field are:

**`0b0000`**
Halting debug is not implemented.

**`0b0010`**
Armv8-M Debug architecture.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [19:0]**
Reserved, RES0.

### D1.2.129 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read-only register located at `0xE000ED60`.

Secure software can access the Non-secure version of this register via ID_ISAR0_NS located at `0xE002ED60`. The location `0xE002ED60` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

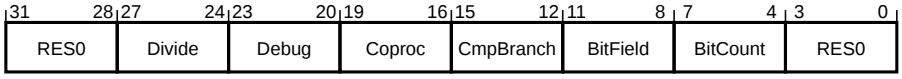
## Preface

If coprocessors excluding the Floating-point Extension are not supported this register reads as `0x01101110`.

If coprocessors excluding the Floating-point Extension are supported this register reads as `0x01141110`.

## Field descriptions

The ID_ISAR0 bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**Divide, bits [27:24]**
Divide. Indicates the supported Divide instructions.

The possible values of this field are:

**`0b0001`**
Supports SDIV and UDIV instructions.

All other values are reserved.

This field reads as `0b0001`.

**Debug, bits [23:20]**
Debug. Indicates the implemented Debug instructions.

The possible values of this field are:

**0b0001**
Supports BKPT instruction.

All other values are reserved.

This field reads as 0b0001.

**Coproc, bits [19:16]**
Coprocessor. Indicates the supported coprocessor instructions.

The possible values of this field are:

**0b0000**
No coprocessor instructions support other than FPU or MVE.

**0b0100**
Coprocessor instructions supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**CmpBranch, bits [15:12]**
Compare and branch. Indicates the supported combined Compare and Branch instructions.

The possible values of this field are:

**0b0001**
Supports CBNZ and CBZ instructions.

**0b0011**
Supports CBNZ and CBZ instructions along with non-predicated low overhead looping (WLS, DLS, LE and LCTP) and branch future (BF, BFX, BFL, BFLX, and BFCSEL) instructions.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**BitField, bits [11:8]**
Bit field. Indicates the supported bit field instructions.

The possible values of this field are:

**0b0001**
BFC, BFI, SBFX, and UBFX supported.

All other values are reserved.

This field reads as 0b0001.

**BitCount, bits [7:4]**
Bit count. Indicates the supported bit count instructions.

The possible values of this field are:

**0b0001**
CLZ supported.

All other values are reserved.

This field reads as 0b0001.

**Bits [3:0]**
Reserved, RES0.

### D1.2.130 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read-only register located at `0xE000ED64`.

Secure software can access the Non-secure version of this register via ID_ISAR1_NS located at `0xE002ED64`. The location `0xE002ED64` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

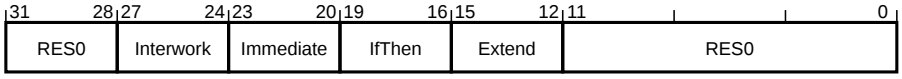
## Preface

If the DSP Extension is not implemented, this register reads as `0x02211000`.

If the DSP Extension is implemented, this register reads as `0x02212000`.

## Field descriptions

The ID_ISAR1 bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**Interwork, bits [27:24]**
Interworking. Indicates the implemented interworking instructions.

The possible values of this field are:

**`0b0010`**
BLX, BX, and loads to PC interwork.

All other values are reserved.

This field reads as `0b0010`.

**Immediate, bits [23:20]**
Immediate. Indicates the implemented for data-processing instructions with long immediates.

The possible values of this field are:

**0b0010**
ADDW, MOVW, MOVT, and SUBW supported.

All other values are reserved.

This field reads as 0b0010.

**IfThen, bits [19:16]**
If-Then. Indicates the implemented If-Then instructions.

The possible values of this field are:

**0b0001**
IT instruction supported.

All other values are reserved.

This field reads as 0b0001.

**Extend, bits [15:12]**
Extend. Indicates the implemented Extend instructions.

The possible values of this field are:

**0b0001**
SXTB, SXTH, UXTB, and UXTH.

**0b0010**
Adds SXTB16, SXTAB, SXTAB16, SXTAH, UXTB16, UXTAB, UXTAB16, and UXTAH, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [11:0]**
Reserved, RES0.

### D1.2.131 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read-only register located at `0xE000ED68`.

Secure software can access the Non-secure version of this register via ID_ISAR2_NS located at `0xE002ED68`. The location `0xE002ED68` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Preface

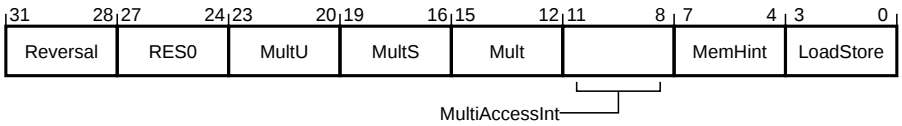
With bits [11:8] masked, if the DSP Extension is not implemented, this register reads as `0x20112032`.

With bits[11:8] masked, if the DSP Extension is implemented, this register reads as `0x20232032`.

The value of bits [11:8] is determined by whether the PE implements restartable or continuable multi-access instructions.

## Field descriptions

The ID_ISAR2 bit assignments are:




**Reversal, bits [31:28]**
Reversal. Indicates the implemented Reversal instructions.

The possible values of this field are:

**`0b0010`**
REV, REV16, REVSH and RBIT instructions supported.

All other values are reserved.

This field reads as `0b0010`.

**Bits [27:24]**
Reserved, RES0.

**MultU, bits [23:20]**
Multiply unsigned. Indicates the implemented advanced unsigned Multiply instructions.

The possible values of this field are:

**0b0001**
UMULL and UMLAL.

**0b0010**
Adds UMAAL, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**MultS, bits [19:16]**
Multiply signed. Indicates the implemented advanced signed Multiply instructions.

The possible values of this field are:

**0b0001**
SMULL and SMLAL.

**0b0011**
Adds all saturating and DSP signed multiplies, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Mult, bits [15:12]**
Multiplies. Indicates the implemented additional Multiply instructions.

The possible values of this field are:

**0b0010**
MUL, MLA, and MLS.

All other values are reserved.

This field reads as `0b0010`.

**MultiAccessInt, bits [11:8]**
Multi-access instructions. Indicates the support for interruptible multi-access instructions.

The possible values of this field are:

**0b0000**
No support. LDM and STM instructions are not interruptible.

**0b0001**
LDM and STM instructions are restartable.

**0b0010**
LDM and STM instructions, and if Armv8.1-M is implemented the CLRM instruction, are continuable.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**MemHint, bits [7:4]**
Memory hints. Indicates the implemented Memory hint instructions.

The possible values of this field are:

**0b0011**
PLI and PLD instructions implemented.

All other values are reserved.

This field reads as `0b0011`.

**LoadStore, bits [3:0]**
Load/store. Indicates the implemented additional load/store instructions.

The possible values of this field are:

**`0b0010`**
Supports load-acquire, store-release, and exclusive load and store instructions.

All other values are reserved.

This field reads as `0b0010`.

### D1.2.132 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read-only register located at `0xE000ED6C`.

Secure software can access the Non-secure version of this register via ID_ISAR3_NS located at `0xE002ED6C`. The location `0xE002ED6C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

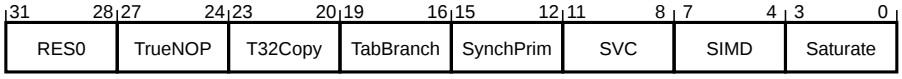
## Preface

If the DSP Extension is not implemented, this register reads as `0x01111110`.

If the DSP Extension is implemented, this register reads as `0x01111131`.

## Field descriptions

The ID_ISAR3 bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**TrueNOP, bits [27:24]**
True no-operation. Indicates the implemented true NOP instructions.

The possible values of this field are:

**`0b0001`**
NOP instruction and compatible hints implemented.

All other values are reserved.

This field reads as `0b0001`.

**T32Copy, bits [23:20]**
T32 copy. Indicates the support for T32 non flag-setting MOV instructions.

The possible values of this field are:

**0b0001**
Encoding T1 of MOV (register) supports copying low register to low register.

All other values are reserved.

This field reads as 0b0001.

**TabBranch, bits [19:16]**
Table branch. Indicates the implemented Table Branch instructions.

The possible values of this field are:

**0b0001**
TBB and TBH implemented.

All other values are reserved.

This field reads as 0b0001.

**SynchPrim, bits [15:12]**
Synchronization primitives. Used in conjunction with ID_ISAR4.SynchPrim_frac to indicate the implemented synchronization primitive instructions.

The possible values of this field are:

**0b0001**
LDREX, STREX, LDREXB, STREXB, LDREXH, STREXH, and CLREX implemented.

All other values are reserved.

This field reads as 0b0001.

**SVC, bits [11:8]**
Supervisor Call. Indicates the implemented SVC instructions.

The possible values of this field are:

**0b0001**
SVC instruction implemented.

All other values are reserved.

This field reads as 0b0001.

**SIMD, bits [7:4]**
Single-instruction, multiple-data. Indicates the implemented SIMD instructions.

The possible values of this field are:

**0b0001**
SSAT, USAT, and Q-bit implemented.

**0b0011**
Adds all packed arithmetic and GE-bits, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Saturate, bits [3:0]**
Saturate. Indicates the implemented saturating instructions.

The possible values of this field are:

**0b0000**
None implemented.

**0b0001**
QADD, QDADD, QDSUB, QSUB, and Q-bit implemented, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.133 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read-only register located at `0xE000ED70`.

Secure software can access the Non-secure version of this register via ID_ISAR4_NS located at `0xE002ED70`. The location `0xE002ED70` is RES0 to software executing in Non-secure state and the debugger.

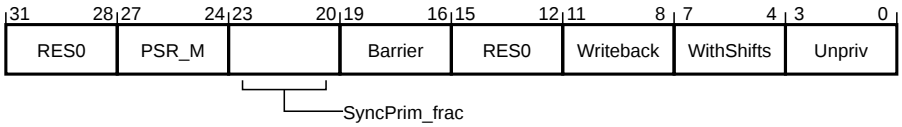
This register is not banked between Security states.

## Preface

This register reads as `0x01310131`.

## Field descriptions

The ID_ISAR4 bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**PSR_M, bits [27:24]**
Program Status Registers M. Indicates the implemented M profile instructions to modify the PSRs.

The possible values of this field are:

**`0b0001`**
M profile forms of CPS, MRS, and MSR implemented.

All other values are reserved.

This field reads as `0b0001`.

**SyncPrim_frac, bits [23:20]**
Synchronization primitives fractional. Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented synchronization primitive instructions.

The possible values of this field are:

**0b0011**
LDREX, STREX, CLREX, LDREXB, LDREXH, STREXB, and STREXH implemented.

All other values are reserved.

This field reads as 0b0011.

**Barrier, bits [19:16]**
Barrier. Indicates the implemented Barrier instructions.

The possible values of this field are:

**0b0001**
CSDB, DMB, DSB, ISB, PSSBB and SSBB barrier instructions implemented.

All other values are reserved.

This field reads as 0b0001.

**Bits [15:12]**
Reserved, RES0.

**Writeback, bits [11:8]**
Writeback. Indicates the support for writeback addressing modes.

The possible values of this field are:

**0b0001**
All writeback addressing modes supported.

All other values are reserved.

This field reads as 0b0001.

**WithShifts, bits [7:4]**
With shifts. Indicates the support for write-back addressing modes.

The possible values of this field are:

**0b0011**
Support for constant shifts on load/store and other instructions.

All other values are reserved.

This field reads as 0b0011.

**Unpriv, bits [3:0]**
Unprivileged. Indicates the implemented unprivileged instructions.

The possible values of this field are:

**0b0010**
LDRBT, LDRHT, LDRSBT, LDRSHT, LDRT, STRBT, STRHT, and STRT implemented.

All other values are reserved.

This field reads as 0b0010.

### D1.2.134 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

**Purpose**
Provides information about the instruction set implemented by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

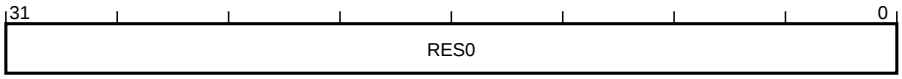
**Attributes**
32-bit read-only register located at `0xE000ED74`.

Secure software can access the Non-secure version of this register via ID_ISAR5_NS located at `0xE002ED74`. The location `0xE002ED74` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_ISAR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.135 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

**Purpose**
Provides information about the implemented memory model and memory management support.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

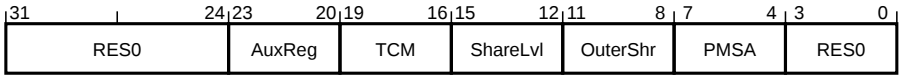
**Attributes**
32-bit read-only register located at `0xE000ED50`.

Secure software can access the Non-secure version of this register via ID_MMFR0_NS located at `0xE002ED50`. The location `0xE002ED50` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_MMFR0 bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**AuxReg, bits [23:20]**
Auxiliary Registers. Indicates support for Auxiliary Control Registers.

The possible values of this field are:

**`0b0000`**
No Auxiliary Control Registers.

**`0b0001`**
Auxiliary Control Registers supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**TCM, bits [19:16]**
Tightly Coupled Memories. Indicates support for Tightly Coupled Memories (TCMs).

The possible values of this field are:

**`0b0000`**
None supported.

**0b0001**
TCMs supported with IMPLEMENTATION DEFINED control.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**ShareLvl, bits [15:12]**
Shareability Levels. Indicates the number of Shareability levels implemented.

The possible values of this field are:

**0b0000**
One level of Shareability implemented.

**0b0001**
Two levels of Shareability implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**OuterShr, bits [11:8]**
Outermost Shareability. Indicates the outermost Shareability domain implemented.

The possible values of this field are:

**0b0000**
Implemented as Non-cacheable.

**0b0001**
Implemented with hardware coherency support.

**0b1111**
Shareability ignored.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**PMSA, bits [7:4]**
Protected memory system architecture. Indicates support for the protected memory system architecture (PMSA).

The possible values of this field are:

**0b0100**
Supports PMSAv8.

All other values are reserved.

This field reads as `0b0100`.

**Bits [3:0]**
Reserved, RES0.

### D1.2.136 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

**Purpose**
Provides information about the implemented memory model and memory management support.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

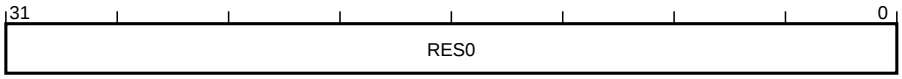
**Attributes**
32-bit read-only register located at `0xE000ED54`.

Secure software can access the Non-secure version of this register via ID_MMFR1_NS located at `0xE002ED54`. The location `0xE002ED54` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_MMFR1 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.137 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

**Purpose**
Provides information about the implemented memory model and memory management support.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

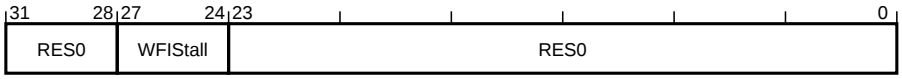
**Attributes**
32-bit read-only register located at `0xE000ED58`.

Secure software can access the Non-secure version of this register via ID_MMFR2_NS located at `0xE002ED58`. The location `0xE002ED58` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_MMFR2 bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**WFIStall, bits [27:24]**
WFI stall. Indicates the support for Wait For Interrupt (WFI) stalling.

The possible values of this field are:

**`0b0000`**
WFI never stalls.

**`0b0001`**
WFI has the ability to stall.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [23:0]**
Reserved, RES0.

## D1.2.138 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

**Purpose**
Provides information about the implemented memory model and memory management support.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

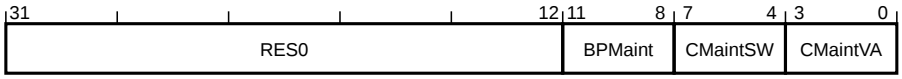
**Attributes**
32-bit read-only register located at `0xE000ED5C`.

Secure software can access the Non-secure version of this register via ID_MMFR3_NS located at `0xE002ED5C`. The location `0xE002ED5C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

### Field descriptions

The ID_MMFR3 bit assignments are:




**Bits [31:12]**
Reserved, RES0.

**BPMaint, bits [11:8]**
Branch predictor maintenance. Indicates the supported branch predictor maintenance.

The possible values of this field are:

**`0b0000`**
None supported.

**`0b0001`**
Support for invalidate all of branch predictors.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**CMaintSW, bits [7:4]**
Cache maintenance set/way. Indicates the supported cache maintenance operations by set/way.

The possible values of this field are:

**`0b0000`**
None supported.

**0b0001**
Maintenance by set/way operations supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**CMaintVA, bits [3:0]**
Cache maintenance by address. Indicates the supported cache maintenance operations by address.

The possible values of this field are:

**0b0000**
None supported.

**0b0001**
Maintenance by address and instruction cache invalidate all supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.139 ID_PFR0, Processor Feature Register 0

The ID_PFR0 characteristics are:

**Purpose**
Gives top-level information about the instruction set supported by the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

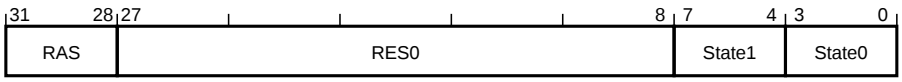
**Attributes**
32-bit read-only register located at `0xE000ED40`.

Secure software can access the Non-secure version of this register via ID_PFR0_NS located at `0xE002ED40`. The location `0xE002ED40` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_PFR0 bit assignments are:




**RAS, bits [31:28]**
RAS Extension. Identifies which version of the RAS extension is implemented.

The possible values of this field are:

**`0b0000`**
No RAS extension.

**`0b0010`**
Version 1 of the RAS extension implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [27:8]**
Reserved, RES0.

**State1, bits [7:4]**
T32 instruction set support.

The possible values of this field are:

**`0b0011`**
T32 instruction set including Thumb-2 Technology implemented.

All other values are reserved.

This field reads as `0b0011`.

**State0, bits [3:0]**

A32 instruction set support.

The possible values of this field are:

**0b0000**

A32 instruction set not implemented.

All other values are reserved.

This field reads as 0b0000.

### D1.2.140 ID_PFR1, Processor Feature Register 1

The ID_PFR1 characteristics are:

**Purpose**
Gives information about the programmers' model and Extensions support.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

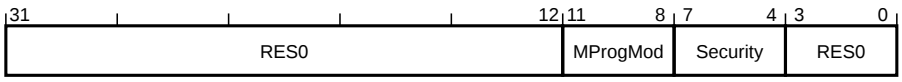
**Attributes**
32-bit read-only register located at `0xE000ED44`.

Secure software can access the Non-secure version of this register via ID_PFR1_NS located at `0xE002ED44`. The location `0xE002ED44` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The ID_PFR1 bit assignments are:




**Bits [31:12]**
Reserved, RES0.

**MProgMod, bits [11:8]**
M programmers' model. Identifies support for the M-Profile programmers' model support.

The possible values of this field are:

**`0b0010`**
Two-stack programmers' model.

All other values are reserved.

This field reads as `0b0010`.

**Security, bits [7:4]**
Security. Identifies whether the Security Extension is implemented.

The possible values of this field are:

**`0b0000`**
Security Extension not implemented.

**`0b0001`**
Security Extension implemented.

**0b0011**
Security Extension implemented with state handling instructions (VSCCLRM, CLRM, FPCXT access instructions and disabling SG thread mode re-entrancy).

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [3:0]**
Reserved, RES0.

### D1.2.141 IPSR, Interrupt Program Status Register

The IPSR characteristics are:

**Purpose**
Provides privileged access to the current exception number field.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

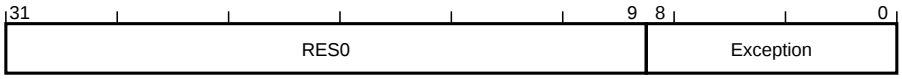
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The IPSR bit assignments are:




**Bits [31:9]**
Reserved, RES0.

**Exception, bits [8:0]**
Exception number. Holds the exception number of the currently-executing exception, or zero for Thread mode.

The possible values of this field are:

**Zero**
PE in Thread mode.

**Non zero**
PE in Handler mode in given exception number. On a function call from Secure state the value is set to 1 to ensure that the Non-secure state cannot determine which exception handler is executing.

This field resets to zero on a Warm reset.

### D1.2.142 ITM_CIDR0, ITM Component Identification Register 0

The ITM_CIDR0 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

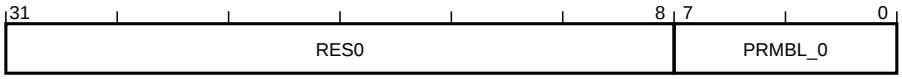
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FF0`.

This register is not banked between Security states.

## Field descriptions

The ITM_CIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_0, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

### D1.2.143 ITM_CIDR1, ITM Component Identification Register 1

The ITM_CIDR1 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

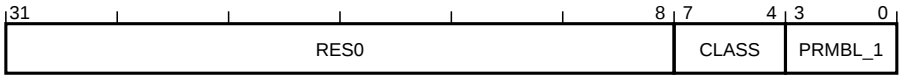
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FF4`.

This register is not banked between Security states.

## Field descriptions

The ITM_CIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

### D1.2.144 ITM_CIDR2, ITM Component Identification Register 2

The ITM_CIDR2 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

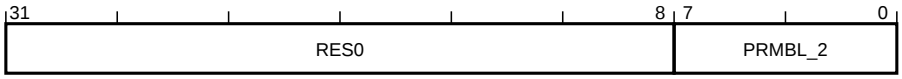
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FF8`.

This register is not banked between Security states.

## Field descriptions

The ITM_CIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

### D1.2.145 ITM_CIDR3, ITM Component Identification Register 3

The ITM_CIDR3 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

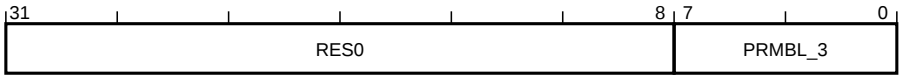
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FFC`.

This register is not banked between Security states.

## Field descriptions

The ITM_CIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

### D1.2.146 ITM_DEVARCH, ITM Device Architecture Register

The ITM_DEVARCH characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

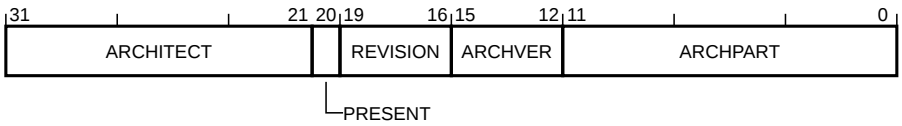
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FBC`.

This register is not banked between Security states.

## Field descriptions

The ITM_DEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**
Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

**`0x23B`**
JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

**PRESENT, bit [20]**
DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

**1**
DEVARCH information present.

This bit reads as one.

**REVISION, bits [19:16]**
Revision. Defines the architecture revision of the component.

The possible values of this field are:

**0b0000**
ITM architecture v2.0.

This field reads as 0b0000.

**ARCHVER, bits [15:12]**
Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

**0b0001**
ITM architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as 0b0001.

**ARCHPART, bits [11:0]**
Architecture Part. Defines the architecture of the component.

The possible values of this field are:

**0xA01**
ITM architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as 0xA01.

### D1.2.147 ITM_DEVTYPE, ITM Device Type Register

The ITM_DEVTYPE characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

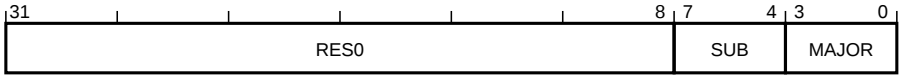
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FCC`.

This register is not banked between Security states.

## Field descriptions

The ITM_DEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Sub-type. Component sub-type.

The possible values of this field are:

**`0x0`**
Other. Only permitted if the MAJOR field reads as `0x0`.

**`0x4`**
Associated with a Bus, stimulus derived from bus activity. Only permitted if the MAJOR field reads as `0x3`.

This field reads as an IMPLEMENTATION DEFINED value.

**MAJOR, bits [3:0]**
Major type. Component major type.

The possible values of this field are:

**0x0**
Miscellaneous.

**0x3**
Trace Source.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.148 ITM_LAR, ITM Software Lock Access Register

The ITM_LAR characteristics are:

**Purpose**

Provides CoreSight Software Lock control for the ITM, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

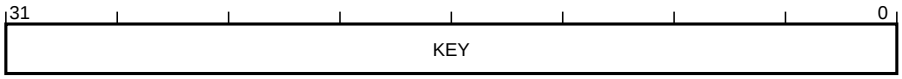
**Attributes**

32-bit write-only register located at `0xE0000FB0`.

This register is not banked between Security states.

## Field descriptions

The ITM_LAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.149 ITM_LSR, ITM Software Lock Status Register

The ITM_LSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the ITM, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

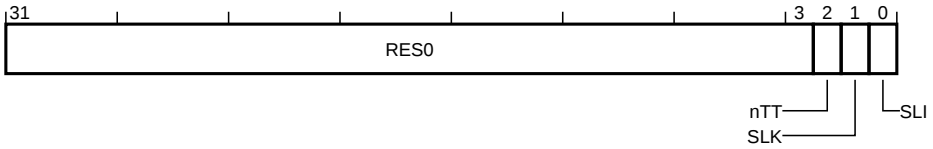
**Attributes**

32-bit read-only register located at `0xE0000FB4`.

This register is not banked between Security states.

## Field descriptions

The ITM_LSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Warm reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

## D1.2.150 ITM_PIDR0, ITM Peripheral Identification Register 0

The ITM_PIDR0 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

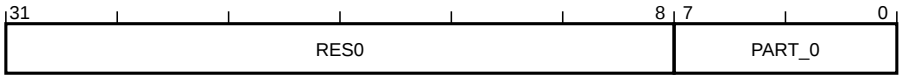
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FE0`.

This register is not banked between Security states.

### Field descriptions

The ITM_PIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.151 ITM_PIDR1, ITM Peripheral Identification Register 1

The ITM_PIDR1 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

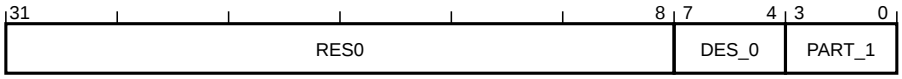
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FE4`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.152 ITM_PIDR2, ITM Peripheral Identification Register 2

The ITM_PIDR2 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

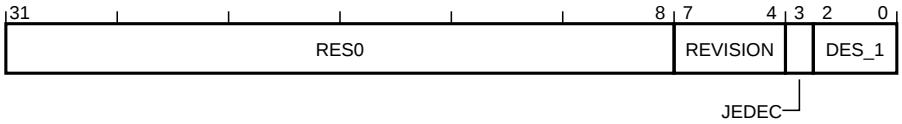
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FE8`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

**DES_1, bits [2:0]**
JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.153 ITM_PIDR3, ITM Peripheral Identification Register 3

The ITM_PIDR3 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

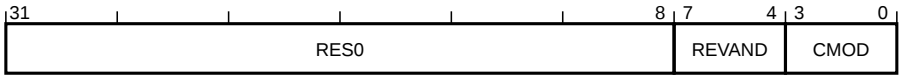
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FEC`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.154 ITM_PIDR4, ITM Peripheral Identification Register 4

The ITM_PIDR4 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

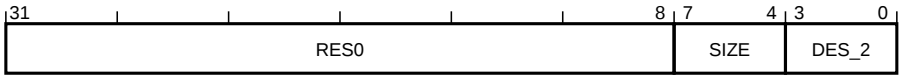
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FD0`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

**DES_2, bits [3:0]**
JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.155 ITM_PIDR5, ITM Peripheral Identification Register 5

The ITM_PIDR5 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

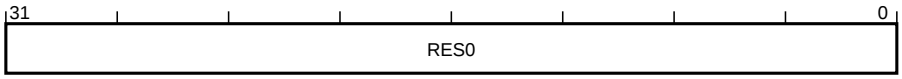
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FD4`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.156 ITM_PIDR6, ITM Peripheral Identification Register 6

The ITM_PIDR6 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

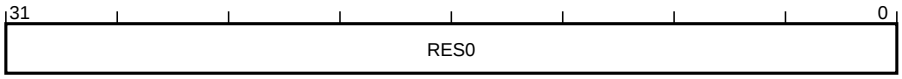
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FD8`.

This register is not banked between Security states.

## Field descriptions

The ITM_PIDR6 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

## D1.2.157 ITM_PIDR7, ITM Peripheral Identification Register 7

The ITM_PIDR7 characteristics are:

**Purpose**
Provides CoreSight discovery information for the ITM.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

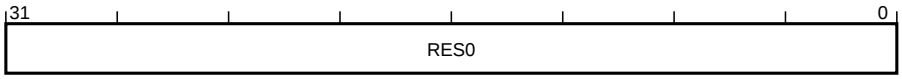
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read-only register located at `0xE0000FDC`.

This register is not banked between Security states.

### Field descriptions

The ITM_PIDR7 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

## D1.2.158 ITM_STIMn, ITM Stimulus Port Register, n = 0 - 255

The ITM_STIM{0..255} characteristics are:

**Purpose**
Provides the interface for generating Instrumentation packets.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored if ITM_TPR.PRIVMASK[$n$ DIV 8] is set to one.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

All writes are ignored if ITM_TCR.ITMENA == 0 or ITM_TER{$n$ DIV 32}.STIMENA[$n$ MOD 32] == 0.

This register is word, halfword, and byte accessible.

Accesses that are not word aligned are UNPREDICTABLE.

**Configurations**
Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read/write register located at `0xE0000000` + 4$n$.

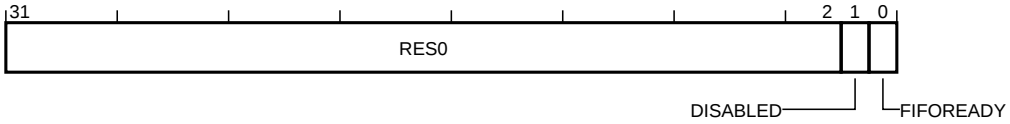
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

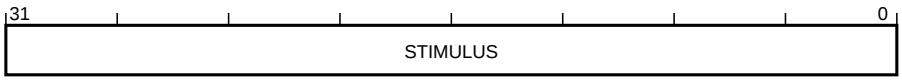
### Field descriptions

The ITM_STIM{0..255} bit assignments are:

On a read:




On a write:




**STIMULUS, bits [31:0], on a write**
Stimulus data. Data to write to the stimulus port output buffer, for forwarding as an Instrumentation packet. The size of write access determines the type of Instrumentation packet generated.

**Bits [31:2], on a read**
Reserved, RES0.

**DISABLED, bit [1], on a read**
Disabled. Indicates whether the stimulus port is enabled or disabled.

The possible values of this bit are:

**0**
Stimulus port and ITM are enabled.

**1**
Stimulus port or ITM is disabled.

**FIFOREADY, bit [0], on a read**
FIFO ready. Indicates whether the stimulus port can accept data.

The possible values of this bit are:

**0**
Stimulus port cannot accept data.

**1**
Stimulus port can accept at least one word.

### D1.2.159 ITM_TCR, ITM Trace Control Register

The ITM_TCR characteristics are:

**Purpose**
Configures and controls transfers through the ITM interface.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

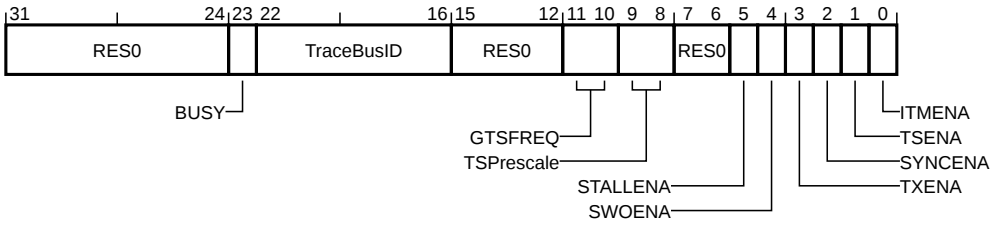
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read/write register located at `0xE0000E80`.

This register is not banked between Security states.

## Field descriptions

The ITM_TCR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**BUSY, bit [23]**
ITM busy. Indicates whether the ITM is currently processing events.

The possible values of this bit are:

**0**
ITM is not processing any events.

**1**
Events present and being drained.

Events means the ITM is generating or processing any of:

- Packets generated by the ITM from writes to Stimulus Ports.
- Other packets generated by the ITM itself.
- Packets generated by the DWT.

This bit is read-only.

**TraceBusID, bits [22:16]**
Trace bus identity. Identifier for multi-source trace stream formatting. If multi-source trace is in use, the debugger must write a unique non-zero trace ID value to this field.

The possible values of this field are:

**0x00**
Multi-source trace not in use.

**0x01-0x6F**
Unique trace ID value to be used for ITM trace packets.

All other values are reserved. If the ITM is the only trace source in the system, this field might be RAZ.

This field resets to an UNKNOWN value on a Cold reset.

**Bits [15:12]**
Reserved, RES0.

**GTSFREQ, bits [11:10]**
Global timestamp frequency. Defines how often the ITM generates a global timestamp, based on the global timestamp clock frequency, or disables generation of global timestamps.

The possible values of this field are:

**0b00**
Disable generation of Global Timestamp packets.

**0b01**
Generate timestamp request whenever the ITM detects a change in global timestamp counter bits [*N*-1:7]. This is approximately every 128 cycles.

**0b10**
Generate timestamp request whenever the ITM detects a change in global timestamp counter bits [*N*-1:13]. This is approximately every 8192 cycles.

**0b11**
Generate a timestamp after every packet, if the output FIFO is empty.

*N* is the size of the global timestamp counter.

If the implementation does not support global timestamping then these bits are reserved, RAZ/WI.

This field resets to zero on a Cold reset.

**TSPrescale, bits [9:8]**
Timestamp prescale. Local timestamp prescaler, used with the trace packet reference clock.

The possible values of this field are:

**0b00**
No prescaling.

**0b01**
Divide by 4.

**0b10**
Divide by 16.

**0b11**
Divide by 64.

If the processor does not implement the timestamp prescaler then these bits are reserved, RAZ/WI.

This field resets to zero on a Cold reset.

**Bits [7:6]**
Reserved, RES0.

**STALLENA, bit [5]**
Stall enable. Stall the PE to guarantee delivery of Data Trace packets.

The possible values of this bit are:

**0**
Drop Hardware Source packets and generate an Overflow packet if the ITM output is stalled.

**1**
Stall the PE to guarantee delivery of Data Trace packets.

If stalling is not implemented, this bit is RAZ/WI.

**SWOENA, bit [4]**
SWO enable. Enables asynchronous clocking of the timestamp counter.

The possible values of this bit are:

**0**
Timestamp counter uses the processor system clock.

**1**
Timestamp counter uses asynchronous clock from the TPIU interface. The timestamp counter is held in reset while the output line is idle.

Which clocking modes are implemented is IMPLEMENTATION DEFINED. If the implementation does not support both modes this bit is either RAZ or RAO, to indicate the implemented mode.

This bit resets to an UNKNOWN value on a Cold reset.

**TXENA, bit [3]**
Transmit enable. Enables forwarding of hardware event packet from the DWT unit to the ITM for output to the TPIU.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

It is IMPLEMENTATION DEFINED whether the DWT discards packets that it cannot forward to the ITM.

This bit resets to zero on a Cold reset.

**Note**

If a debugger changes this bit from 0 to 1, the DWT might forward a hardware event packet that it has previously generated.

**SYNCENA, bit [2]**
Synchronization enable. Enables Synchronization packet transmission for a synchronous TPIU.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

This bit resets to zero on a Cold reset.

**Note**

If a debugger sets this bit to 1 it must also configure DWT_CTRL.SYNCTAP for the correct synchronization speed.

**TSENA, bit [1]**
Timestamp enable. Enables Local timestamp generation.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

This bit resets to zero on a Cold reset.

**ITMENA, bit [0]**
ITM enable. Enables the ITM.

The possible values of this bit are:

**0**
Disabled.

**1**
Enabled.

This is the master enable for the ITM unit. A debugger must set this bit to 1 to permit writes to all Stimulus Port registers.

This bit resets to zero on a Cold reset.

## D1.2.160 ITM_TERn, ITM Trace Enable Register, n = 0 - 7

The ITM_TER{0..7} characteristics are:

**Purpose**
Provide an individual enable bit for each ITM_STIM register.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

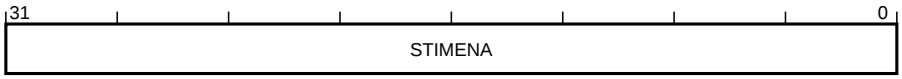
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read/write register located at `0xE0000E00` + 4*n*.

This register is not banked between Security states.

### Field descriptions

The ITM_TER{0..7} bit assignments are:




**STIMENA, bits [31:0]**
Stimulus enable. For STIMENA[*m*] in ITM_TER*n*, controls whether stimulus port ITM_STIM<32*n*+*m*> is enabled.

The possible values of each bit are:

**0**
Stimulus port (32*n* + *m*) disabled.

**1**
Stimulus port (32*n* + *m*) enabled.

Bits corresponding to unimplemented stimulus ports are RAZ/WI. Unprivileged writes to ITM_TER*n* do not update STIMENA[*m*] if ITM_TPR.PRIVMASK[(32*n*+*m*) DIV 8] is set to 1.

This field resets to zero on a Cold reset.

### D1.2.161 ITM_TPR, ITM Trace Privilege Register

The ITM_TPR characteristics are:

**Purpose**
Controls which stimulus ports can be accessed by unprivileged code.

**Usage constraints**
If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.

If the Main Extension is not implemented, unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the ITM is implemented.

This register is RES0 if the ITM is not implemented.

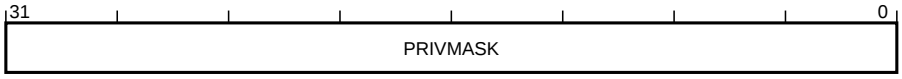
If the Main Extension is not implemented then the ITM is not implemented.

**Attributes**
32-bit read/write register located at `0xE0000E40`.

This register is not banked between Security states.

## Field descriptions

The ITM_TPR bit assignments are:




**PRIVMASK, bits [31:0]**
Privilege mask. For PRIVMASK[*m*], defines the access permissions of stimulus ports ITM_STIM<8*m*> to ITM_STIM<8*m*+7> inclusive.

The possible values of each bit are:

**0**
Unprivileged access permitted.

**1**
Privileged access only.

Bits corresponding to unimplemented stimulus ports are RAZ/WI.

This field resets to zero on a Cold reset.

## D1.2.162 LO_BRANCH_INFO, Loop and branch tracking information

The LO_BRANCH_INFO characteristics are:

**Purpose**
Holds the cached loop end point and branching information.

**Usage constraints**
This register is not accessible from software.

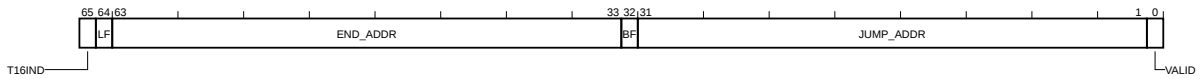
**Configurations**
This register is always implemented.

**Attributes**
66-bit read/write register.

This register is not banked between Security states.

### Field descriptions

The LO_BRANCH_INFO bit assignments are:




**T16IND, bit [65]**
When set this field indicates that BF is a T16 indirect branch. For BF and link instructions, this flag calculates the offset of the return address set in LR from the branch point.

**LF, bit [64]**
Link / forever. If BF is set, this field indicates that the link register is populated with a return address at the point the branch is taken. If BF is clear, this flag indicates a forever loop that does not decrement LR at the end of each loop iteration.

**END_ADDR, bits [63:33]**
The partial address of either the last instruction in a low-overhead-loop, or an upcomming branch set by a Branch Future instruction. If the VALID bit is set and this field matches the address of the next instruction, a branch back to the start of the loop is triggered (as specified by the JUMP_ADDR field).

**BF, bit [32]**
Indicates that the value in this register originates from a BF instruction.

**JUMP_ADDR, bits [31:1]**
The address to jump to when an end address match is detected.

**VALID, bit [0]**
The cached loop information in the rest of this register is only valid if this bit is set. The PE is permitted to clear this bit and invalidate the cache at any point.

This bit resets to zero on a Warm reset.

### D1.2.163 LR, Link Register

The LR characteristics are:

**Purpose**
Exception and procedure call link register.

**Usage constraints**
Privileged and unprivileged access permitted.

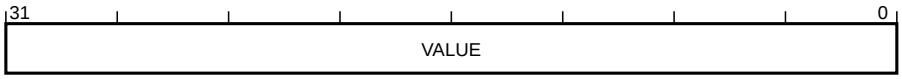
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The LR bit assignments are:




**VALUE, bits [31:0]**
Link register. 32-bit link register updated to hold a return address, FNC_RETURN or EXC_RETURN on a function call or exception entry. LR can be used as a general-purpose register.

This field resets to an UNKNOWN value on Warm reset when the Main Extension is not implemented.

This field resets to `0xFFFFFFFF` on a Warm reset if the Main Extension is implemented.

### D1.2.164 MAIR_ATTR, Memory Attribute Indirection Register Attributes

The MAIR_ATTR characteristics are:

**Purpose**
Defines the memory attribute encoding for use in the MPU_MAIR0 and MPU_MAIR1.

**Usage constraints**
None.

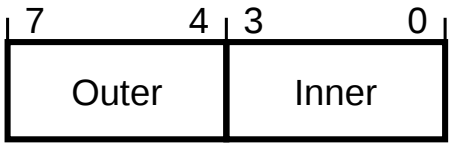
**Configurations**
All.

**Attributes**
8-bit payload.

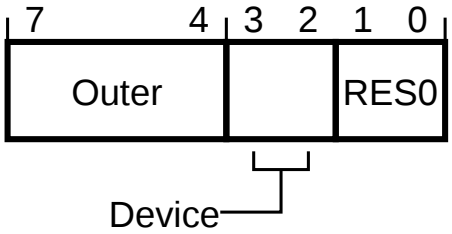
## Field descriptions

The MAIR_ATTR bit assignments are:

**When Outer != 0b0000**:




**When Outer == 0b0000**:




**Outer, bits [7:4]**
Outer attributes. Specifies the Outer memory attributes.

The possible values of this field are:

**0b0000**
Device memory.

**0b00RW**
Normal memory, Outer Write-Through transient (RW!=0b00).

**0b0100**
Normal memory, Outer Non-cacheable.

**0b01RW**
Normal memory, Outer Write-Back Transient (RW!=0b00).

**0b10RW**
Normal memory, Outer Write-Through Non-transient.

**0b11RW**
Normal memory, Outer Write-Back Non-transient.

R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

**Device, bits [3:2], when Outer == 0b0000**
Device attributes. Specifies the memory attributes for Device.

The possible values of this field are:

**0b00**
Device-nGnRnE.

**0b01**
Device-nGnRE.

**0b10**
Device-nGRE.

**0b11**
Device-GRE.

**Bits [1:0] , when Outer == 0b0000]**
Reserved, RES0.

**Inner, bits [3:0], when Outer != 0b0000**
Inner attributes. Specifies the Inner memory attributes.

The possible values of this field are:

**0b0000**
UNPREDICTABLE.

**0b00RW**
Normal memory, Inner Write-Through Transient (RW!=0b00).

**0b0100**
Normal memory, Inner Non-cacheable.

**0b01RW**
Normal memory, Inner Write-Back Transient (RW!=0b00).

**0b10RW**
Normal memory, Inner Write-Through Non-transient.

**0b11RW**
Normal memory, Inner Write-Back Non-transient.

R and W specify the inner read and write allocation policy: 0 = do not allocate, 1 = allocate.

### D1.2.165 MMFAR, MemManage Fault Address Register

The MMFAR characteristics are:

**Purpose**
Shows the address of the memory location that caused an MPU fault.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000ED34`.

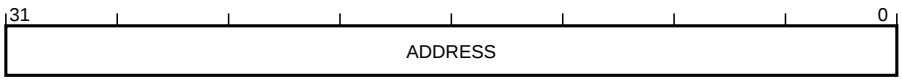
Secure software can access the Non-secure version of this register via MMFAR_NS located at `0xE002ED34`. The location `0xE002ED34` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The MMFAR bit assignments are:




**ADDRESS, bits [31:0]**
Data address for an MemManage fault. This register is updated with the address of a location that produced a MemManage fault. The MMFSR shows the cause of the fault, and whether this field is valid. This field is valid only when MMFSR.MMARVALID is set, otherwise it is UNKNOWN.

In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if BFSR.BFARVALID is set.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.166 MMFSR, MemManage Fault Status Register

The MMFSR characteristics are:

**Purpose**
Shows the status of MPU faults.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
8-bit read/write-one-to-clear register located at `0xE000ED28`.

Secure software can access the Non-secure version of this register via MMFSR_NS located at `0xE002ED28`. The location `0xE002ED28` is RES0 to software executing in Non-secure state and the debugger.

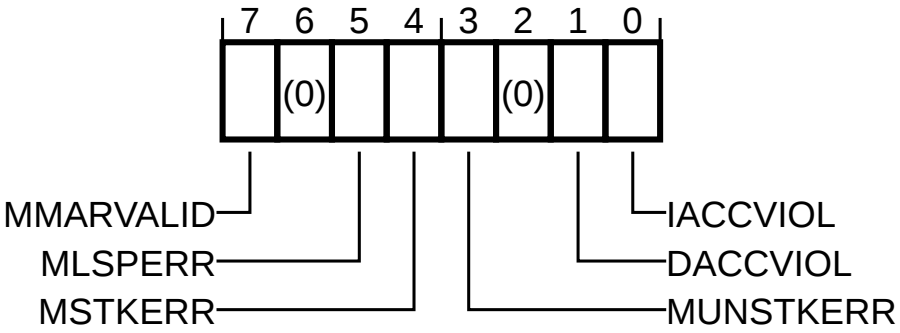
This register is banked between Security states.

This register is part of CFSR.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The MMFSR bit assignments are:




**MMARVALID, bit [7]**
MMFAR valid flag. Indicates validity of the MMFAR register.

The possible values of this bit are:

**0**
MMFAR content not valid.

**1**
MMFAR content valid.

This bit resets to zero on a Warm reset.

**Bit [6]**
Reserved, RES0.

**MLSPERR, bit [5]**
MemManage lazy state preservation error flag. Records whether a MemManage fault occurred during FP lazy state preservation.

The possible values of this bit are:

**0**
No MemManage occurred.

**1**
MemManage occurred.

This bit resets to zero on a Warm reset.

**MSTKERR, bit [4]**
MemManage stacking error flag. Records whether a derived MemManage fault occurred during exception entry stacking.

The possible values of this bit are:

**0**
No derived MemManage occurred.

**1**
Derived MemManage occurred during exception entry.

This bit resets to zero on a Warm reset.

**MUNSTKERR, bit [3]**
MemManage unstacking error flag. Records whether a derived MemManage fault occurred during exception return unstacking.

The possible values of this bit are:

**0**
No derived MemManage fault occurred.

**1**
Derived MemManage fault occurred during exception return.

This bit resets to zero on a Warm reset.

**Bit [2]**
Reserved, RES0.

**DACCVIOL, bit [1]**
Data access violation flag. Records whether a data access violation has occurred.

The possible values of this bit are:

**0**
No MemManage fault on data access has occurred.

**1**
MemManage fault on data access has occurred.

A DACCVIOL will be accompanied by an MMFAR update.

This bit resets to zero on a Warm reset.

**IACCVIOL, bit [0]**
Instruction access violation. Records whether an instruction related memory access violation has occurred.

The possible values of this bit are:

**0**
No MemManage fault on instruction access has occurred.

**1**
MemManage fault on instruction access has occurred.

An IACCVIOL is only recorded if a faulted instruction is executed.

This bit resets to zero on a Warm reset.

## D1.2.167 MPU_CTRL, MPU Control Register

The MPU_CTRL characteristics are:

**Purpose**

Enables the MPU and, when the MPU is enabled, controls whether the default memory map is enabled as a background region for privileged accesses, and whether the MPU is enabled for HardFaults, NMIs, and exception handlers when FAULTMASK is set to 1.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

This register is always implemented.

**Attributes**

32-bit read/write register located at `0xE000ED94`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

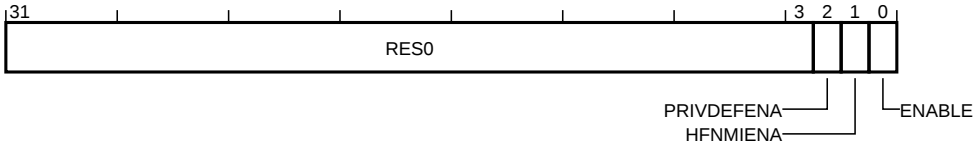
Secure software can access the Non-secure version of this register via MPU_CTRL_NS located at `0xE002ED94`. The location `0xE002ED94` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The MPU_CTRL bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**PRIVDEFENA, bit [2]**

Privileged default enable. Controls whether the default memory map is enabled for privileged software.

The possible values of this bit are:

**0**

Use of default memory map disabled.

**1**

Use of default memory map enabled for privilege code.

When the ENABLE bit is set to 0, the PE ignores the PRIVDEFENA bit. If no regions are enabled and the PRIVDEFENA and ENABLE bits are set to 1, only privileged code can execute from the system address map. If no MPU regions are implemented this bit is RES0.

This bit resets to zero on a Warm reset.

**HFNMIENA, bit [1]**

HardFault, NMI enable. Controls whether handlers executing with priority less than 0 access memory with the MPU enabled or disabled. This applies to HardFaults and NMIs when FAULTMASK is set to 1.

The possible values of this bit are:

**0**

MPU disabled for these handlers.

**1**

MPU enabled for these handlers.

If HFNMIENA is set to 1 when ENABLE is set to 0, behavior is UNPREDICTABLE. If no MPU regions are implemented this bit is RES0.

This bit resets to zero on a Warm reset.

**ENABLE, bit [0]**

Enable. Enables the MPU.

The possible values of this bit are:

**0**

The MPU is disabled.

**1**

The MPU is enabled.

Disabling the MPU, by setting the ENABLE bit to 0, means that privileged and unprivileged accesses use the default memory map. If no MPU regions are implemented this bit is RES0.

This bit resets to zero on a Warm reset.

## D1.2.168 MPU_MAIR0, MPU Memory Attribute Indirection Register 0

The MPU_MAIR0 characteristics are:

**Purpose**
Along with MPU_MAIR1, provides the memory attribute encodings corresponding to the AttrIndx values.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDC0`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_MAIR0_NS located at `0xE002EDC0`. The location `0xE002EDC0` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

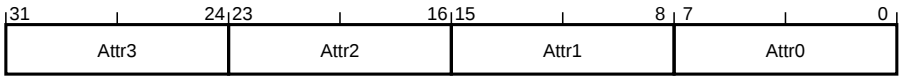
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

### Field descriptions

The MPU_MAIR0 bit assignments are:




**Attr*m*, bits [8*m*+7:8*m*], for *m* = 0 to 3**
Attribute *m*. Memory attribute encoding for MPU regions with an AttrIndx of *m*.

The possible values of this field are:

**All**
See MAIR_ATTR for encoding.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.169 MPU_MAIR1, MPU Memory Attribute Indirection Register 1

The MPU_MAIR1 characteristics are:

**Purpose**
Along with MPU_MAIR0, provides the memory attribute encodings corresponding to the AttrIndx values.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDC4`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_MAIR1_NS located at `0xE002EDC4`. The location `0xE002EDC4` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

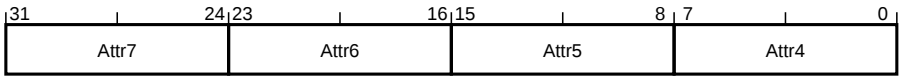
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

## Field descriptions

The MPU_MAIR1 bit assignments are:




**Attr*m*, bits [8(*m*-4)+7:8(*m*-4)], for *m* = 4 to 7**
Attribute *m*. Memory attribute encoding for MPU regions with an AttrIndx of *m*.

The possible values of this field are:

**All**
See MAIR_ATTR for encoding.

This field resets to an UNKNOWN value on a Warm reset.

## D1.2.170 MPU_RBAR, MPU Region Base Address Register

The MPU_RBAR characteristics are:

**Purpose**
Provides indirect read and write access to the base address of the currently selected MPU region for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED9C`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_RBAR_NS located at `0xE002ED9C`. The location `0xE002ED9C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

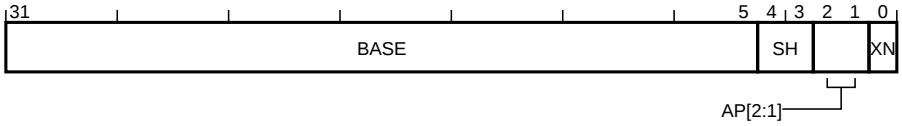
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Preface

This register provides access to the configuration of the MPU region selected by MPU_RNR.REGION for the appropriate Security state. The field descriptions apply to the currently selected region.

### Field descriptions

The MPU_RBAR bit assignments are:




**BASE, bits [31:5]**
Base address. Contains bits [31:5] of the lower inclusive limit of the selected MPU memory region. This value is zero extended to provide the base address to be checked against.

This field resets to an UNKNOWN value on a Warm reset.

**SH, bits [4:3]**
Shareability. Defines the Shareability domain of this region for Normal memory.

The possible values of this field are:

**`0b00`**
Non-shareable.

**`0b10`**
Outer Shareable.

**0b11**
Inner Shareable.

All other values are reserved.

For any type of Device memory, the value of this field is ignored.

This field resets to an UNKNOWN value on a Warm reset.

**AP[2:1], bits [2:1]**
Access permissions. Defines the access permissions for this region.

The possible values of this field are:

**0b00**
Read/write by privileged code only.

**0b01**
Read/write by any privilege level.

**0b10**
Read-only by privileged code only.

**0b11**
Read-only by any privilege level.

This field resets to an UNKNOWN value on a Warm reset.

**XN, bit [0]**
Execute Never. Defines whether code can be executed from this region.

The possible values of this bit are:

**0**
Execution only permitted if read permitted.

**1**
Execution not permitted.

This bit resets to an UNKNOWN value on a Warm reset.

## D1.2.171 MPU_RBAR_An, MPU Region Base Address Register Alias, n = 1 - 3

The MPU_RBAR_A{1..3} characteristics are:

**Purpose**

Provides indirect read and write access to the base address of the MPU region selected by MPU_RNR[7:2]:(*n*[1:0]) for the selected Security state.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read/write register located at `0xE000EDA4` + 8(*n*-1).

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_RBAR_An_NS located at `0xE002EDA4` + 8(n-1). The location `0xE002EDA4` + 8(n-1) is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

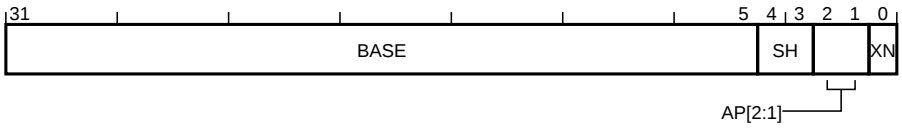
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Preface

This register is an alias of the MPU_RBAR register and provides access to the configuration of the MPU region selected by MPU_RNR.REGION had REGION[1:0] been set to *n*[1:0].

### Field descriptions

The MPU_RBAR_A{1..3} bit assignments are:




**BASE, bits [31:5]**

Base address. Contains bits [31:5] of the lower inclusive limit of the selected MPU memory region. This value is zero extended to provide the base address to be checked against.

This field resets to an UNKNOWN value on a Warm reset.

**SH, bits [4:3]**

Shareability. Defines the Shareability domain of this region for Normal memory.

The possible values of this field are:

**`0b00`**

Non-shareable.

**`0b10`**

Outer Shareable.

**0b11**
Inner Shareable.

All other values are reserved.

For any type of Device memory, the value of this field is ignored.

This field resets to an UNKNOWN value on a Warm reset.

**AP[2:1], bits [2:1]**
Access permissions. Defines the access permissions for this region.

The possible values of this field are:

**0b00**
Read/write by privileged code only.

**0b01**
Read/write by any privilege level.

**0b10**
Read-only by privileged code only.

**0b11**
Read-only by any privilege level.

This field resets to an UNKNOWN value on a Warm reset.

**XN, bit [0]**
Execute Never. Defines whether code can be executed from this region.

The possible values of this bit are:

**0**
Execution only permitted if read permitted.

**1**
Execution not permitted.

This bit resets to an UNKNOWN value on a Warm reset.

### D1.2.172 MPU_RLAR, MPU Region Limit Address Register

The MPU_RLAR characteristics are:

**Purpose**
Provides indirect read and write access to the limit address of the currently selected MPU region for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDA0`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_RLAR_NS located at `0xE002EDA0`. The location `0xE002EDA0` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

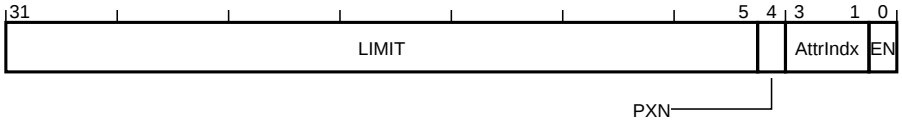
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Preface

This register provides access to the configuration of the MPU region selected by MPU_RNR.REGION for the appropriate Security state. The field descriptions apply to the currently selected region.

## Field descriptions

The MPU_RLAR bit assignments are:




**LIMIT, bits [31:5]**
Limit address. Contains bits [31:5] of the upper inclusive limit of the selected MPU memory region. This value is postfixed with `0x1F` to provide the limit address to be checked against.

This field resets to an UNKNOWN value on a Warm reset.

**PXN, bit [4]**
Privileged execute never. Defines whether code can be executed from this privileged region.

The possible values of this bit are:

**0**
Execution only permitted if read permitted.

**1**
Execution from a privileged mode is not permitted.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**AttrIndx, bits [3:1]**
Attribute index. Associates a set of attributes in the MPU_MAIR0 and MPU_MAIR1 fields.

This field resets to an UNKNOWN value on a Warm reset.

**EN, bit [0]**
Enable. Region enable.

The possible values of this bit are:

**0**
Region disabled.

**1**
Region enabled.

This bit resets to zero on a Warm reset.

### D1.2.173 MPU_RLAR_An, MPU Region Limit Address Register Alias, n = 1 - 3

The MPU_RLAR_A{1..3} characteristics are:

**Purpose**

Provides indirect read and write access to the limit address of the currently selected MPU region selected by MPU_RNR[7:2]:(*n*[1:0]) for the selected Security state.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read/write register located at `0xE000EDA8` + 8(*n*-1).

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via MPU_RLAR_An_NS located at `0xE002EDA8` + 8(n-1). The location `0xE002EDA8` + 8(n-1) is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

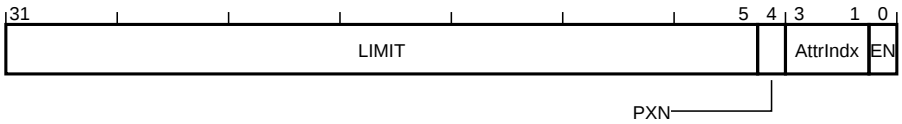
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Preface

This register is an alias of the MPU_RLAR register and provides access to the configuration of the MPU region selected by MPU_RNR.REGION had REGION[1:0] been set to *n*[1:0].

## Field descriptions

The MPU_RLAR_A{1..3} bit assignments are:




**LIMIT, bits [31:5]**

Limit address. Contains bits [31:5] of the upper inclusive limit of the selected MPU memory region. This value is postfixed with `0x1F` to provide the limit address to be checked against.

This field resets to an UNKNOWN value on a Warm reset.

**PXN, bit [4]**

Privileged execute never. Defines whether code can be executed from this privileged region.

The possible values of this bit are:

**0**

Execution only permitted if read permitted.

**1**

Execution from a privileged mode is not permitted.

If version Armv8.1-M of the architecture is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**AttrIndx, bits [3:1]**
Attribute index. Associates a set of attributes in the MPU_MAIR0 and MPU_MAIR1 fields.

This field resets to an UNKNOWN value on a Warm reset.

**EN, bit [0]**
Enable. Region enable.

The possible values of this bit are:

**0**
Region disabled.

**1**
Region enabled.

This bit resets to zero on a Warm reset.

### D1.2.174 MPU_RNR, MPU Region Number Register

The MPU_RNR characteristics are:

**Purpose**
Selects the region currently accessed by MPU_RBAR and MPU_RLAR.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED98`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

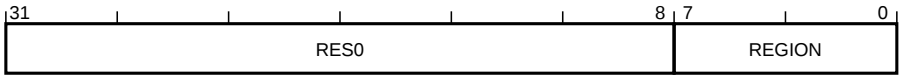
Secure software can access the Non-secure version of this register via MPU_RNR_NS located at `0xE002ED98`. The location `0xE002ED98` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The MPU_RNR bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REGION, bits [7:0]**
Region number. Indicates the memory region accessed by MPU_RBAR and MPU_RLAR.

If no MPU regions are implemented, this field is RES0. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.175 MPU_TYPE, MPU Type Register

The MPU_TYPE characteristics are:

**Purpose**
The MPU Type Register indicates how many regions the MPU for the selected Security state supports.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

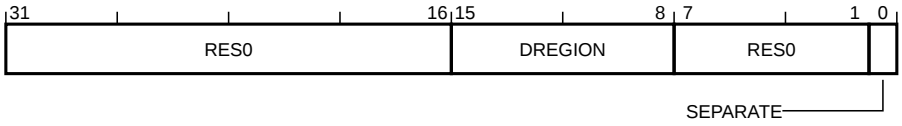
**Attributes**
32-bit read-only register located at `0xE000ED90`.

Secure software can access the Non-secure version of this register via MPU_TYPE_NS located at `0xE002ED90`. The location `0xE002ED90` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

## Field descriptions

The MPU_TYPE bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**DREGION, bits [15:8]**
Data regions. Number of regions supported by the MPU.

If this field reads-as-zero, the PE does not implement an MPU for the selected Security state.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [7:1]**
Reserved, RES0.

**SEPARATE, bit [0]**
Separate. Indicates support for separate instructions and data address regions.

Armv8-M only supports unified MPU regions.

This bit reads as zero.

### D1.2.176 MSPLIM, Main Stack Pointer Limit Register

The MSPLIM characteristics are:

**Purpose**
Holds the lower limit of the Main stack pointer.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

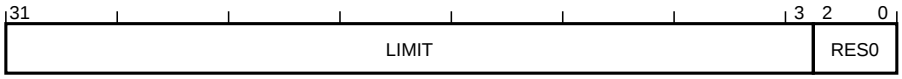
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The MSPLIM bit assignments are:




**LIMIT, bits [31:3]**
Stack limit. Bits [31:3] of the Main stack pointer limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure MSPLIM is RES0.

This field resets to zero on a Warm reset.

**Bits [2:0]**
Reserved, RES0.

### D1.2.177 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

**Purpose**
Describes the features provided by the Floating-point Extension.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

**Attributes**
32-bit read-only register located at `0xE000EF40`.

Secure software can access the Non-secure version of this register via MVFR0_NS located at `0xE002EF40`. The location `0xE002EF40` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Preface

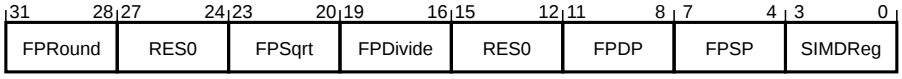
When the Floating-point Extension is not implemented this register reads as `0x00000000`.

Where single-precision only Floating-point is supported this register reads as `0x10110021`.

Where single and double-precision Floating-point are supported this register reads as `0x10110221`.

## Field descriptions

The MVFR0 bit assignments are:




**FPRound, bits [31:28]**
Floating-point rounding modes. Indicates the rounding modes supported by the Floating-point Extension.

The possible values of this field are:

**`0b0000`**
Not supported.

**`0b0001`**
All rounding modes supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [27:24]**
Reserved, RES0.

**FPSqrt, bits [23:20]**
Floating-point square root. Indicates the support for Floating-point square root operations.

The possible values of this field are:

**0b0000**
Not supported.

**0b0001**
Supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FPDivide, bits [19:16]**
Floating-point divide. Indicates the support for Floating-point divide operations.

The possible values of this field are:

**0b0000**
Not supported.

**0b0001**
Supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [15:12]**
Reserved, RES0.

**FPDP, bits [11:8]**
Floating-point double-precision. Indicates support for Floating-point double-precision operations.

The possible values of this field are:

**0b0000**
Not supported.

**0b0010**
Supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FPSP, bits [7:4]**
Floating-point single-precision. Indicates support for Floating-point single-precision operations.

The possible values of this field are:

**0b0000**
Not supported.

**0b0010**
Supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**SIMDReg, bits [3:0]**
SIMD registers. Indicates size of Floating-Point Extension register file.

The possible values of this field are:

**0b0001**
16 x 64-bit registers.

All other values are reserved.

This field reads as `0b0001`.

### D1.2.178 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

**Purpose**
Describes the features provided by the Floating-point Extension.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

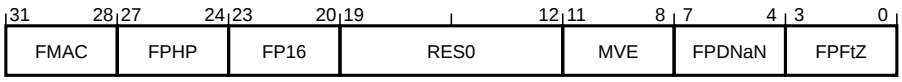
**Attributes**
32-bit read-only register located at `0xE000EF44`.

Secure software can access the Non-secure version of this register via MVFR1_NS located at `0xE002EF44`. The location `0xE002EF44` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The MVFR1 bit assignments are:




**FMAC, bits [31:28]**
Fused multiply accumulate. Indicates whether the Floaing-point Extension implements the fused multiply accumulate instructions.

The possible values of this field are:

**`0b0000`**
Not supported.

**`0b0001`**
Implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FPHP, bits [27:24]**
Floating-point half-precision conversion. Indicates whether the Floating-point Extension implements half-precision Floating-point conversion instructions.

The possible values of this field are:

**`0b0000`**
Not supported.

**`0b0001`**
Half-precision to single-precision implemented.

**0b0010**
Half-precision to single and double-precision implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FP16, bits [23:20]**
Floating-point half-precision data processing. Indicates whether the FP Extension implements half-precision FP data processing instructions.

The possible values of this field are:

**0b0000**
No Half-precision data processing support.

**0b0001**
Half-precision data processing instructions supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [19:12]**
Reserved, RES0.

**MVE, bits [11:8]**
Indicates support for M-profile vector extension.

The possible values of this field are:

**0b0000**
Not supported.

**0b0001**
Supported, no Floating-point.

**0b0010**
Supported, with single-precision and half-precision Floating-point.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FPDNaN, bits [7:4]**
Floating-point default NaN. Indicates whether the Floating-point Extension implementation supports NaN propagation.

The possible values of this field are:

**0b0000**
Not supported.

**0b0001**
Propagation of NaN values supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**FPFtZ, bits [3:0]**
Floating-point flush-to-zero. Indicates whether subnormals are always flushed-to-zero.

The possible values of this field are:

**0b0000**
Not supported.

**0b0001**
Full denormalized numbers arithmetic supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.179 MVFR2, Media and VFP Feature Register 2

The MVFR2 characteristics are:

**Purpose**
Describes the features provided by the Floating-point Extension.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present if the Floating-point Extension, or MVE, or both are implemented.

This register is RES0 if neither the Floating-point Extension nor MVE are implemented.

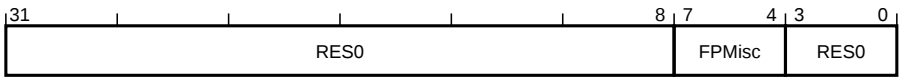
**Attributes**
32-bit read-only register located at `0xE000EF48`.

Secure software can access the Non-secure version of this register via MVFR2_NS located at `0xE002EF48`. The location `0xE002EF48` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The MVFR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**FPMisc, bits [7:4]**
Floating-point miscellaneous. Indicates support for miscellaneous FP features.

The possible values of this field are:

**`0b0000`**
Floating-point extensions not implemented.

**`0b0100`**
Selection, directed conversion to integer, VMINNM and VMAXNM supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [3:0]**
Reserved, RES0.

## D1.2.180 NSACR, Non-secure Access Control Register

The NSACR characteristics are:

**Purpose**

Defines the Non-secure access permissions for the Floating-point Extension and coprocessors CP0 to CP7. If MVE is implemented this register Specifies the Non-secure access permissions for MVE.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**

32-bit read/write register located at `0xE000ED8C`.

If the Security Extension is not implemented this register returns a value of `0x00000CFF`.

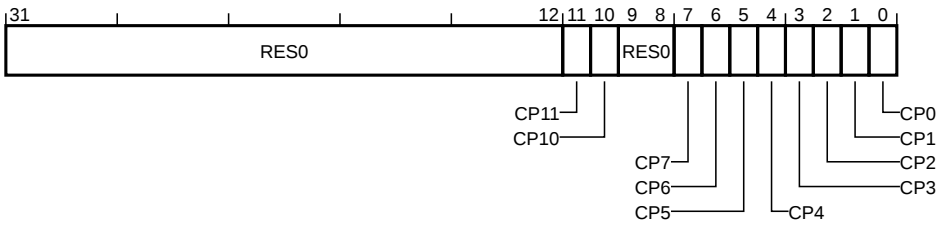
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The NSACR bit assignments are:




**Bits [31:12]**

Reserved, RES0.

**CP11, bit [11]**

CP11 access. Enables Non-secure access to the Floating-point Extension and MVE.

Programming with a different value than that used for CP10 is UNPREDICTABLE.

If the Floating-point Extension and MVE are not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**CP10, bit [10]**

CP10 access. Enables Non-secure access to the Floating-point Extension and MVE.

The possible values of this bit are:

**0**

Non-secure accesses to the Floating-point Extension or MVE, unless otherwise specified, generate a NOCP UsageFault.

**1**
Non-secure access to the Floating-point Extension or MVE permitted.

If the Floating-point Extension and MVE are not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [9:8]**
Reserved, RES0.

**CP*m*, bit [*m*], for *m* = 0 to 7**
CP*m* access. Enables Non-secure access to coprocessor CP*m*.

The possible values of this field are:

**0**
Non-secure accesses to this coprocessor generate a NOCP UsageFault.

**1**
Non-secure access to this coprocessor permitted.

A CP*m* bit is RAZ/WI if CP*m* is:

- Not implemented.
- Not enabled for the Security state in which the PE is executing.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.181 NVIC_IABRn, Interrupt Active Bit Register, n = 0 - 15

The NVIC_IABR{0..15} characteristics are:

**Purpose**
For each group of 32 interrupts, shows the active state of each interrupt.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

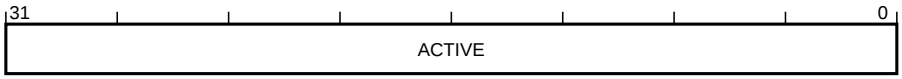
**Attributes**
32-bit read-only register located at `0xE000E300` + 4*n*.

Secure software can access the Non-secure version of this register via NVIC_IABRn_NS located at `0xE002E300` + 4n. The location `0xE002E300` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC_IABR{0..15} bit assignments are:




**ACTIVE, bits [31:0]**
Active state. For ACTIVE[*m*] in NVIC_IABR*n*, indicates the active state for interrupt 32*n*+*m*.

The possible values of each bit are:

**0**
Interrupt not active.

**1**
Interrupt is active.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.182 NVIC_ICERn, Interrupt Clear Enable Register, n = 0 - 15

The NVIC_ICER{0..15} characteristics are:

**Purpose**
Clears or reads the enabled state of each group of 32 interrupts.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

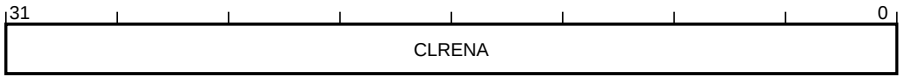
**Attributes**
32-bit read/write-one-to-clear register located at `0xE000E180` + 4*n*.

Secure software can access the Non-secure version of this register via NVIC_ICERn_NS located at `0xE002E180` + 4n. The location `0xE002E180` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC_ICER{0..15} bit assignments are:




**CLRENA, bits [31:0], on a write**
Clear enable. For CLRENA[*m*] in NVIC_ICER*n*, allows interrupt 32*n* + *m* to be disabled.

The possible values of each bit are:

**0**
No effect.

**1**
Disable interrupt 32*n* + *m*.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

**CLRENA, bits [31:0], on a read**
Clear enable. For CLRENA[*m*] in NVIC_ICER*n*, indicates whether interrupt 32*n* + *m* is enabled.

The possible values of each bit are:

**0**
Interrupt 32*n* + *m* disabled.

**1**
Interrupt 32*n* + *m* enabled.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.183 NVIC_ICPRn, Interrupt Clear Pending Register, n = 0 - 15

The NVIC_ICPR{0..15} characteristics are:

**Purpose**
Clears or reads the pending state of each group of 32 interrupts.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

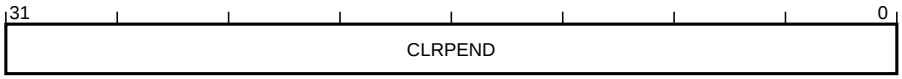
**Attributes**
32-bit read/write-one-to-clear register located at `0xE000E280` + 4*n*.

Secure software can access the Non-secure version of this register via NVIC_ICPRn_NS located at `0xE002E280` + 4n. The location `0xE002E280` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC_ICPR{0..15} bit assignments are:




**CLRPEND, bits [31:0], on a write**
Clear pending. For CLRPEND[*m*] in NVIC_ICPR*n*, allows interrupt 32*n* + *m* to be unpended.

The possible values of each bit are:

**0**
No effect.

**1**
Clear pending state of interrupt 32*n* + *m*.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

**CLRPEND, bits [31:0], on a read**
Clear pending. For CLRPEND[*m*] in NVIC_ICPR*n*, indicates whether interrupt 32*n* + *m* is pending.

The possible values of each bit are:

**0**
Interrupt 32*n* + *m* is not pending.

**1**
Interrupt 32*n* + *m* is pending.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

## D1.2.184 NVIC_IPRn, Interrupt Priority Register, n = 0 - 123

The NVIC_IPR{0..123} characteristics are:

**Purpose**
Sets or reads interrupt priorities.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000E400` + 4*n*.

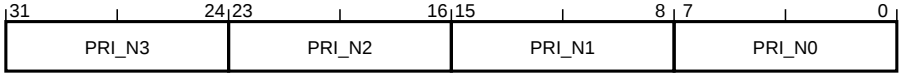
Secure software can access the Non-secure version of this register via NVIC_IPRn_NS located at `0xE002E400` + 4n. The location `0xE002E400` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The NVIC_IPR{0..123} bit assignments are:




**PRI_N*m*, bits [8*m*+7:8*m*], for *m* = 0 to 3**
Priority 'N'+*m*. For register NVIC_IPR*n*, this field indicates and allows modification of the priority of interrupt number 4*n*+*m*, or is RES0 if the PE does not implement this interrupt.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

If interrupt number 4*n*+*m* targets Secure state, this field is RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.185 NVIC_ISERn, Interrupt Set Enable Register, n = 0 - 15

The NVIC_ISER{0..15} characteristics are:

**Purpose**
Enables or reads the enabled state of each group of 32 interrupts.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

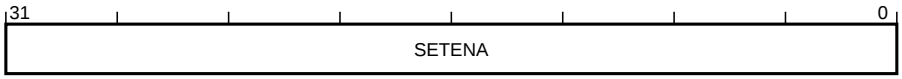
**Attributes**
32-bit read/write-one-to-set register located at `0xE000E100` + 4*n*.

Secure software can access the Non-secure version of this register via NVIC_ISERn_NS located at `0xE002E100` + 4n. The location `0xE002E100` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC_ISER{0..15} bit assignments are:




**SETENA, bits [31:0], on a write**
Set enable. For SETENA[*m*] in NVIC_ISER*n*, allows interrupt 32*n* + *m* to be set enabled.

The possible values of each bit are:

**0**
No effect.

**1**
Enable interrupt 32*n* + *m*.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

**SETENA, bits [31:0], on a read**
Set enable. For SETENA[*m*] in NVIC_ISER*n*, indicates whether interrupt 32*n* + *m* is enabled.

The possible values of each bit are:

**0**
Interrupt 32*n* + *m* disabled.

**1**
Interrupt 32*n* + *m* enabled.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.186 NVIC_ISPRn, Interrupt Set Pending Register, n = 0 - 15

The NVIC_ISPR{0..15} characteristics are:

**Purpose**
Enables or reads the pending state of each group of 32 interrupts.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

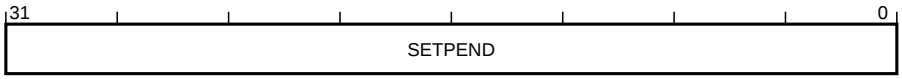
**Attributes**
32-bit read/write-one-to-set register located at `0xE000E200` + 4*n*.

Secure software can access the Non-secure version of this register via NVIC_ISPRn_NS located at `0xE002E200` + 4n. The location `0xE002E200` + 4n is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The NVIC_ISPR{0..15} bit assignments are:




**SETPEND, bits [31:0], on a write**
Set pending. For SETPEND[*m*] in NVIC_ISPR*n*, allows interrupt 32*n* + *m* to be set pending.

The possible values of each bit are:

**0**
No effect.

**1**
Pend interrupt 32*n* + *m*.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field is write-one-to-set. Writes of zero are ignored.

This field resets to zero on a Warm reset.

**SETPEND, bits [31:0], on a read**
Set pending. For SETPEND[*m*] in NVIC_ISPR*n*, indicates whether interrupt 32*n* + *m* is pending.

The possible values of each bit are:

**0**
Interrupt 32*n* + *m* is not pending.

**1**
Interrupt 32*n* + *m* pending.

Bits corresponding to unimplemented interrupts are RES0. Bits corresponding to interrupts targeting Secure state are RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

### D1.2.187 NVIC_ITNSn, Interrupt Target Non-secure Register, n = 0 - 15

The NVIC_ITNS{0..15} characteristics are:

**Purpose**
For each group of 32 interrupts, determines whether each interrupt targets Non-secure or Secure state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000E380` + 4*n*.

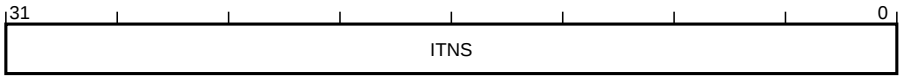
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The NVIC_ITNS{0..15} bit assignments are:




**ITNS, bits [31:0]**
Interrupt Targets Non-secure. For ITNS[*m*] in NVIC_ITNS*n*, this field indicates and allows modification of the target Security state for interrupt 32*n*+*m*.

The possible values of each bit are:

**0**
Interrupt targets Secure state.

**1**
Interrupt targets Non-secure state.

Bits corresponding to unimplemented interrupts are RES0. It is IMPLEMENTATION DEFINED whether individual bits are WI and have an IMPLEMENTATION DEFINED constant value. Where an interrupt is configured to target Secure state, accesses to the associated fields in Non-secure versions of the NVIC_IABR, NVIC_ICER, NVIC_ISER, NVIC_ICPR, NVIC_IPR and NVIC_ISPR are RAZ/WI.

This field resets to zero on a Warm reset.

### D1.2.188 PC, Program Counter

The PC characteristics are:

**Purpose**
Holds the current Program Counter value.

**Usage constraints**
Privileged and unprivileged access permitted.

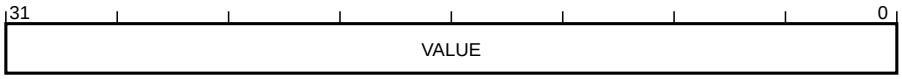
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The PC bit assignments are:




**VALUE, bits [31:0]**
Program Counter. Holds the address of the current instruction.

Software can refer to PC as R15.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.189 PMU_AUTHSTATUS, Performance Monitoring Unit Authentication Status Register

The PMU_AUTHSTATUS characteristics are:

**Purpose**

Provides information about the state of the IMPLEMENTATION DEFINED authentication interface for Performance Monitoring Units.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

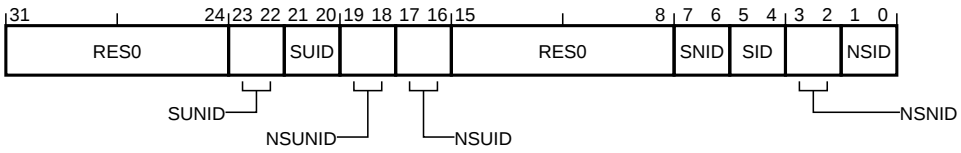
**Attributes**

32-bit read-only register located at `0xE0003FB8`.

This register is not banked between Security states.

## Field descriptions

The PMU_AUTHSTATUS bit assignments are:




**Bits [31:24]**

Reserved, RES0.

**SUNID, bits [23:22]**

Secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive debug is allowed for the Secure state.

The possible values of this field are:

**`0b00`**

Security Extension or Unprivileged Non-invasive Debug not implemented.

**`0b01`**

Reserved.

**`0b10`**

Secure Non-invasive debug prohibited or not restricted to an unprivileged mode.

**`0b11`**

Secure Non-invasive debug allowed only for an unprivileged mode.

If UDE is not implemented, this field is RES0.

**SUID, bits [21:20]**
Secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Halting Debug is allowed for the Secure state.

The possible values of this field are:

**0b00**
Security Extension or Unprivileged Debug not implemented.

**0b01**
Reserved.

**0b10**
Secure halting debug prohibited or not restricted to an unprivileged mode.

**0b11**
Secure halting debug allowed only for an unprivileged mode.

This reflects the value of UnprivHaltingDebugAllowed(TRUE) && !SecureHaltingDebugAllowed().

If UDE is not implemented, this field is RES0.

**NSUNID, bits [19:18]**
Non-secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive Debug is allowed for the Non-secure state.

The possible values of this field are:

**0b00**
Unprivileged Non-invasive debug not implemented.

**0b01**
Reserved.

**0b10**
Non-secure Non-invasive debug prohibited or not restricted to an unprivileged mode.

**0b11**
Non-secure Non-invasive debug allowed only for an unprivileged mode.

If the Main Extension is not implemented, this field is RES0.

**NSUID, bits [17:16]**
Non-secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Halting Debug is allowed for the Non-secure state.

The possible values of this field are:

**0b00**
Unprivileged halting debug not implemented.

**0b01**
Reserved.

**0b10**
Non-secure halting debug prohibited or not restricted to an unprivileged mode.

**0b11**
Non-secure halting debug allowed only for an unprivileged mode.

This reflects the value of UnprivHaltingDebugAllowed(FALSE) && !HaltingDebugAllowed().

If UDE is not implemented, this field is RES0.

**Bits [15:8]**
Reserved, RES0.

**SNID, bits [7:6]**
Secure Non-invasive Debug. Indicates whether Secure non-invasive debug is implemented and allowed.

The possible values of this field are:

**0b00**
Security Extension not implemented.

**0b01**
Reserved.

**0b10**
Security Extension implemented and Secure non-invasive debug prohibited.

**0b11**
Security Extension implemented and Secure non-invasive debug allowed.

**SID, bits [5:4]**
Secure Invasive Debug. Indicates whether Secure invasive debug is implemented and allowed.

The possible values of this field are:

**0b00**
Security Extension not implemented.

**0b01**
Reserved.

**0b10**
Security Extension implemented and Secure invasive debug prohibited.

**0b11**
Security Extension implemented and Secure invasive debug allowed.

**NSNID, bits [3:2]**
Non-secure Non-invasive Debug. Indicates whether Non-secure non-invasive debug is allowed.

The possible values of this field are:

**0b0x**
Reserved.

**0b10**
Non-secure non-invasive debug prohibited.

**0b11**
Non-secure non-invasive debug allowed.

**NSID, bits [1:0]**
Non-secure Invasive Debug. Indicates whether Non-secure invasive debug is allowed.

The possible values of this field are:

**0b0x**
Reserved.

**0b10**
Non-secure invasive debug prohibited.

**0b11**
Non-secure invasive debug allowed.

### D1.2.190 PMU_CCFILTR, Performance Monitoring Unit Cycle Counter Filter Register

The PMU_CCFILTR characteristics are:

**Purpose**
This register is reserved for future use.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

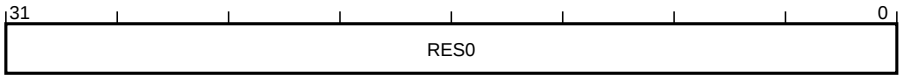
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read/write register located at `0xE000347C`.

This register is not banked between Security states.

## Field descriptions

The PMU_CCFILTR bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.191 PMU_CCNTR, Performance Monitoring Unit Cycle Counter Register

The PMU_CCNTR characteristics are:

**Purpose**
Holds the value of the cycle counter, which counts processor clock cycles.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read/write register located at `0xE000307C`.

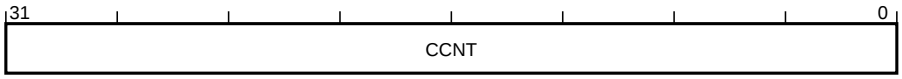
This register is not banked between Security states.

## Preface

This register is an alias of the DWT_CYCCNT register.

## Field descriptions

The PMU_CCNTR bit assignments are:




**CCNT, bits [31:0]**
Cycle count. The cycle count increments on every processor clock cycle.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.192 PMU_CIDR0, Performance Monitoring Unit Component Identification Register 0

The PMU_CIDR0 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

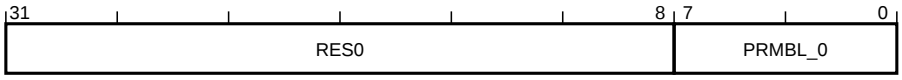
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FF0`.

This register is not banked between Security states.

## Field descriptions

The PMU_CIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_0, bits [7:0]**
Preamble.

This field reads as `0x0D`.

### D1.2.193 PMU_CIDR1, Performance Monitoring Unit Component Identification Register 1

The PMU_CIDR1 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

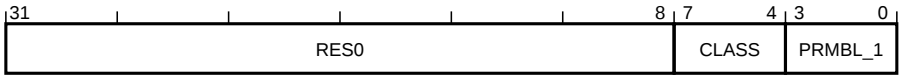
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FF4`.

This register is not banked between Security states.

## Field descriptions

The PMU_CIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
Component class.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
Preamble.

This field reads as `0x0`.

### D1.2.194 PMU_CIDR2, Performance Monitoring Unit Component Identification Register 2

The PMU_CIDR2 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

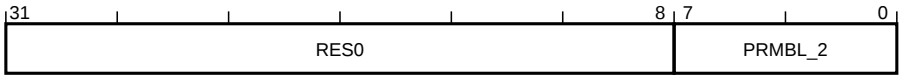
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FF8`.

This register is not banked between Security states.

## Field descriptions

The PMU_CIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
Preamble.

This field reads as `0x05`.

### D1.2.195 PMU_CIDR3, Performance Monitoring Unit Component Identification Register 3

The PMU_CIDR3 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

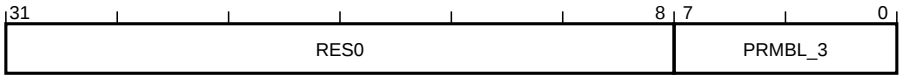
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FFC`.

This register is not banked between Security states.

## Field descriptions

The PMU_CIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
Preamble.

This field reads as `0xB1`.

### D1.2.196 PMU_CNTENCLR, Performance Monitoring Unit Count Enable Clear Register

The PMU_CNTENCLR characteristics are:

**Purpose**

Disables the Cycle Count Register, PMU_CCNTR, and any implemented event counters PMU_EVCNTR<n>. Reading this register shows which counters are enabled.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

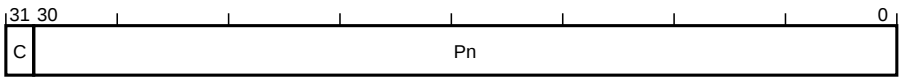
**Attributes**

32-bit read/write register located at `0xE0003C20`.

This register is not banked between Security states.

## Field descriptions

The PMU_CNTENCLR bit assignments are:




**C, bit [31]**

PMU_CCNTR disable bit. Disables the cycle counter register.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter is disabled. When written, has no effect.

**`0b1`**

When read, means the cycle counter is enabled. When written, disables the cycle counter.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter PMU_EVCNTR<n> disable bit. Disables PMU_EVCNTR<n>.

The possible values of this field are:

**`0b0`**

When read, means that PMU_EVCNTR<n> is disabled. When written, has no effect.

**`0b1`**

When read, means that PMU_EVCNTR<n> event counter is enabled. When written, disables PMU_EVCNTR<n>.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.197 PMU_CNTENSET, Performance Monitoring Unit Count Enable Set Register

The PMU_CNTENSET characteristics are:

**Purpose**

Enables the Cycle Count Register, PMU_CCNTR, and any implemented event counters PMU_EVCNTR<n>. Reading this register shows which counters are enabled.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

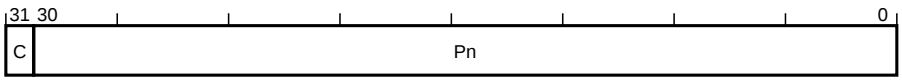
**Attributes**

32-bit read/write register located at `0xE0003C00`.

This register is not banked between Security states.

## Field descriptions

The PMU_CNTENSET bit assignments are:




**C, bit [31]**

PMU_CCNTR enable bit. Enables the cycle counter register.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter is disabled. When written, has no effect.

**`0b1`**

When read, means the cycle counter is enabled. When written, enables the cycle counter.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter PMU_EVCNTR<n> enable bit. Enables PMU_EVCNTR<n>.

The possible values of this field are:

**`0b0`**

When read, means that PMU_EVCNTR<n> is disabled. When written, has no effect.

**`0b1`**

When read, means that PMU_EVCNTR<n> event counter is enabled. When written, enables PMU_EVCNTR<n>.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.198 PMU_CTRL, Performance Monitoring Unit Control Register

The PMU_CTRL characteristics are:

**Purpose**
Configures and controls the Performance Monitoring Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

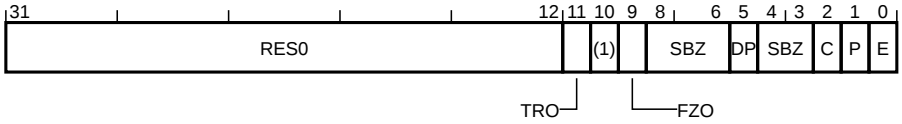
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read/write register located at `0xE0003E04`.

This register is not banked between Security states.

## Field descriptions

The PMU_CTRL bit assignments are:




**Bits [31:12]**
Reserved, RES0.

**TRO, bit [11]**
Trace-on-overflow. Enable trace-on-overflow.

The possible values of this bit are:

**`0b0`**
Trace-on-overflow disabled.

**`0b1`**
Trace whenever any of the first eight counters overflows an 8-bit value.

This bit resets to an UNKNOWN value on a Warm reset.

**Bit [10]**
SBZ.

**FZO, bit [9]**
Freeze-on-overflow. Stops events being counted once PMU_OVSCLR or PMU_OVSSET is non-zero.

The possible values of this bit are:

**`0b0`**
This bit has no effect on event counting.

**0b1**
While PMU_OVSCLR or PMU_OVSSET is nonzero, event counters do not count events.

This bit resets to an UNKNOWN value on a Warm reset.

**Bits [8:6]**
SBZ.

**DP, bit [5]**
Disable cycle counter when event counting is prohibited. This bit is an alias of the DWT_CTRL.CYCDISS bit.

**Bits [4:3]**
SBZ.

**C, bit [2]**
Cycle counter reset. Reset the PMU_CCNTR counter.

The possible values of this bit are:

**0b0**
No action.

**0b1**
Reset PMU_CCNTR to zero.

Resetting PMU_CCNTR does not clear the PMU_CCNTR overflow bit to 0.

This bit is write-only.

This bit reads as zero.

**P, bit [1]**
Event counter reset. Reset event counters.

The possible values of this bit are:

**0b0**
No action.

**0b1**
Reset all event counters, not including PMU_CCNTR, to zero.

Resetting the event counters does not clear any overflow bits to 0.

This bit is write-only.

This bit reads as zero.

**E, bit [0]**
Enable. Enable the event counters.

The possible values of this bit are:

**0b0**
All counters, including PMU_CCNTR, are disabled.

**0b1**
All counters are enabled by PMU_CNTENSET.

This bit resets to zero on a Warm reset.

### D1.2.199 PMU_DEVARCH, Performance Monitoring Unit Device Architecture Register

The PMU_DEVARCH characteristics are:

**Purpose**
Identifies the programmers' model architecture of the Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

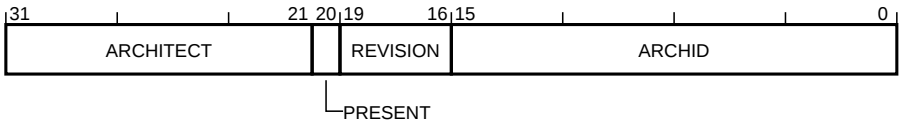
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FBC`.

This register is not banked between Security states.

## Field descriptions

The PMU_DEVARCH bit assignments are:




**ARCHITECT, bits [31:21]**
Defines the architecture of the component.

For Performance Monitoring Units, this is Arm Limited.

Bits [31:28] are the JEP 106 continuation code, `0x4`.

Bits [27:21] are the JEP 106 ID code, `0x3B`.

**PRESENT, bit [20]**
Determines the presence of DEVARCH. When set to 1, indicates that the DEVARCH is present.

This bit reads as `0x1`.

**REVISION, bits [19:16]**
Defines the architecture revision.

For architectures defined by Arm this is the minor revision.

For Performance Monitoring Units, the revision defined by Armv8.1-M is `0x0`.

All other values are reserved.

**ARCHID, bits [15:0]**
Defines this part to be an ARMv8-M debug component.

For architectures defined by Arm this is further subdivided. For Performance Monitoring Units:

Bits [15:12] are the architecture version, `0x0`.

Bits [11:0] are the architecture part number, `0xA06`.

## D1.2.200 PMU_DEVTYPE, Performance Monitoring Unit Device Type Register

The PMU_DEVTYPE characteristics are:

**Purpose**
Indicates to a debugger that this component is part of the Performance Monitoring Unit interface of the PE.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

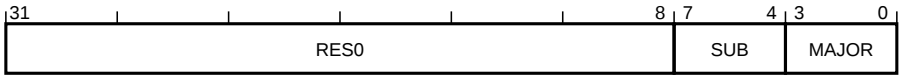
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FCC`.

This register is not banked between Security states.

### Field descriptions

The PMU_DEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Subtype.

This field reads as `0x1`.

**MAJOR, bits [3:0]**
Major type.

This field reads as `0x6`.

## D1.2.201 PMU_EVCNTRn, Performance Monitoring Unit Event Counter Register

The PMU_EVCNTR{0..30} characteristics are:

**Purpose**
Holds performance counter n, which counts events.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read/write register located at `0xE0003000` + 4*n*.

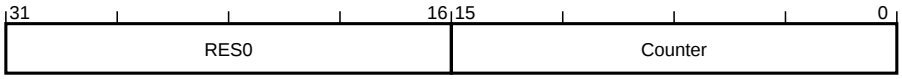
This register is not banked between Security states.

### Preface

If n is greater than or equal to the number of accessible counters, reads and writes of this register are RES0.

### Field descriptions

The PMU_EVCNTR{0..30} bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**Counter, bits [15:0]**
Event counter n.

Value of event counter n, where n is the number of this register. n is a number in the range 0-30. The size of this counter is 16 bits.

The counter counts whenever the selected event occurs, and either of: .

- SecureNoninvasiveDebugAllowed() == TRUE .
- the NS-Req for the operation is set to Non-secure and NoninvasiveDebugAllowed() == TRUE.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.202 PMU_EVTYPERn, Performance Monitoring Unit Event Type and Filter Register

The PMU_EVTYPER{0..30} characteristics are:

**Purpose**
Configures event counter n, where n is 0 to 30.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read/write register located at `0xE0003400` + 4*n*.

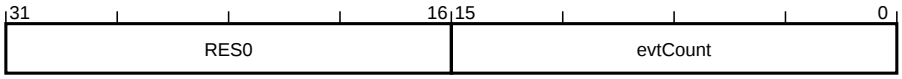
This register is not banked between Security states.

## Preface

If n is greater than or equal to the number of accessible counters, reads and writes of this register are RES0.

## Field descriptions

The PMU_EVTYPER{0..30} bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**evtCount, bits [15:0]**
Event to Count. The event number of the event that is counted by event counter PMU_EVCNTR<n>. If the associated counter does not support the event number that is written to this register, the value read back is UNKNOWN.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.203 PMU_INTENCLR, Performance Monitoring Unit Interrupt Enable Clear Register

The PMU_INTENCLR characteristics are:

**Purpose**

Disables the generation of interrupt requests on overflows from the Cycle Count Register, PMU_CCNTR, and the event counters, PMU_EVCNTR. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

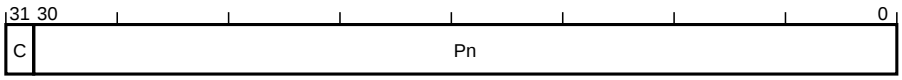
**Attributes**

32-bit read/write register located at `0xE0003C60`.

This register is not banked between Security states.

## Field descriptions

The PMU_INTENCLR bit assignments are:




**C, bit [31]**

PMU_CCNTR overflow interrupt request disable bit. Disable the overflow interrupt for the cycle counter.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

**`0b1`**

When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter overflow interrupt request disable bit for PMU_EVCNTR<n>. Disable the overflow interrupt for PMU_EVCNTR<n>.

The possible values of this field are:

**`0b0`**

When read, means that the PMU_EVCNTR<n> event counter interrupt request is disabled. When written, has no effect.

**0b1**

When read, means that the PMU_EVCNTR<n> event counter interrupt request is enabled. When written, disables the PMU_EVCNTR<n> interrupt request.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.204 PMU_INTENSET, Performance Monitoring Unit Interrupt Enable Set Register

The PMU_INTENSET characteristics are:

**Purpose**

Enables the generation of interrupt requests on overflows from the Cycle Count Register, PMU_CCNTR, and the event counter, PMU_EVCNTR. Reading the register shows which overflow interrupt requests are enabled.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

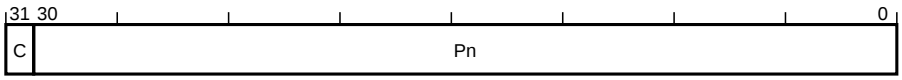
**Attributes**

32-bit read/write register located at `0xE0003C40`.

This register is not banked between Security states.

## Field descriptions

The PMU_INTENSET bit assignments are:




**C, bit [31]**

PMU_CCNTR overflow interrupt request enable bit. Enable the overflow interrupt for the cycle counter.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect.

**`0b1`**

When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter overflow interrupt request enable bit for PMU_EVCNTR<n>. Enable the overflow interrupt for PMU_EVCNTR<n>.

The possible values of this field are:

**`0b0`**

When read, means that the PMU_EVCNTR<n> event counter interrupt request is disabled. When written, has no effect.

**0b1**

When read, means that the PMU_EVCNTR<n> event counter interrupt request is enabled. When written, enables the PMU_EVCNTR<n> interrupt request.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.205 PMU_OVSCLR, Performance Monitoring Unit Overflow Flag Status Clear Register

The PMU_OVSCLR characteristics are:

**Purpose**

Contains the state of the overflow bit for the Cycle Count Register, PMU_CCNTR, and each of the implemented event counters, PMU_EVCNTR<n>. Writing to this register clears these bits.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

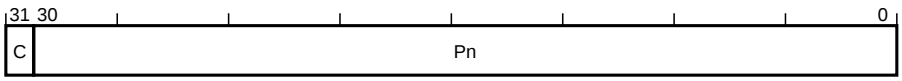
**Attributes**

32-bit read/write register located at `0xE0003C80`.

This register is not banked between Security states.

## Field descriptions

The PMU_OVSCLR bit assignments are:




**C, bit [31]**

PMU_CCNTR overflow bit. Clears the PMU_CCNTR overflow bit.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter has not overflowed. When written, has no effect.

**`0b1`**

When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter overflow clear bit for PMU_EVCNTR<n>. Clears the PMU_EVCNTR<n> overflow bit.

The possible values of this field are:

**`0b0`**

When read, means that the PMU_EVCNTR<n> event counter has not overflowed. When written, has no effect.

**`0b1`**

When read, means that the PMU_EVCNTR<n> event counter has overflowed. When written, clears the PMU_EVCNTR<n> overflow bit to 0.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.206 PMU_OVSSET, Performance Monitoring Unit Overflow Flag Status Set Register

The PMU_OVSSET characteristics are:

**Purpose**

Sets the state of the overflow bit for the Cycle Count Register, PMU_CCNTR, and each of the implemented event counters, PMU_EVCNTR<n>.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**

Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

This register is RES0 if the PMU is not implemented.

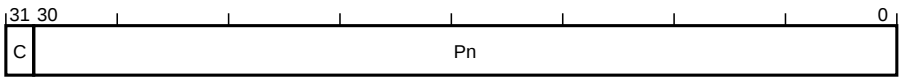
**Attributes**

32-bit read/write register located at `0xE0003CC0`.

This register is not banked between Security states.

## Field descriptions

The PMU_OVSSET bit assignments are:




**C, bit [31]**

PMU_CCNTR overflow bit. Set the overflow status for PMU_CCNTR.

The possible values of this bit are:

**`0b0`**

When read, means the cycle counter has not overflowed. When written, has no effect.

**`0b1`**

When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.

This bit resets to zero on a Cold reset.

**Pn, bits [30:0]**

Event counter overflow set bit for PMU_EVCNTR<n>. Set the overflow status for PMU_EVCNTR<n>.

The possible values of this field are:

**`0b0`**

When read, means that the PMU_EVCNTR<n> event counter has not overflowed. When written, has no effect.

**`0b1`**

When read, means that the PMU_EVCNTR<n> event counter has overflowed. When written, sets the PMU_EVCNTR<n> overflow bit to 1.

This field resets to zero on a Cold reset.

**Note**

Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

### D1.2.207 PMU_PIDR0, Performance Monitoring Unit Peripheral Identification Register 0

The PMU_PIDR0 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

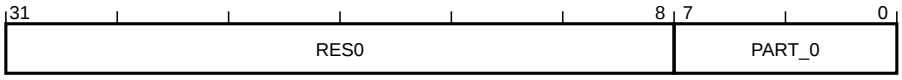
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FE0`.

This register is not banked between Security states.

## Field descriptions

The PMU_PIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number, least significant byte.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.208 PMU_PIDR1, Performance Monitoring Unit Peripheral Identification Register 1

The PMU_PIDR1 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

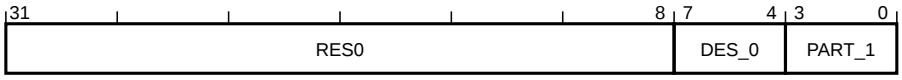
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FE4`.

This register is not banked between Security states.

## Field descriptions

The PMU_PIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
Designer, least significant nibble of JEP106 ID code. For Arm Limited, this field is `0b1011`.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number, most significant nibble.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.209 PMU_PIDR2, Performance Monitoring Unit Peripheral Identification Register 2

The PMU_PIDR2 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

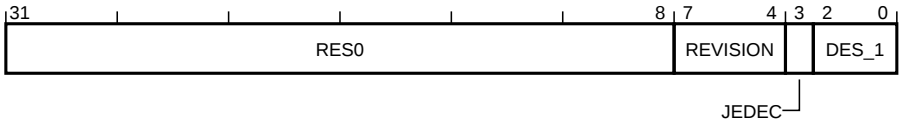
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FE8`.

This register is not banked between Security states.

## Field descriptions

The PMU_PIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Part major revision. Parts can also use this field to extend Part number to 16-bits.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC. RAO. Indicates a JEP106 identity code is used.

**DES_1, bits [2:0]**
Designer, most significant bits of JEP106 ID code. For Arm Limited, this field is `0b011`.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.210 PMU_PIDR3, Performance Monitoring Unit Peripheral Identification Register 3

The PMU_PIDR3 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

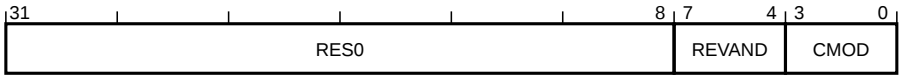
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FEC`.

This register is not banked between Security states.

## Field descriptions

The PMU_PIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
Part minor revision. Parts using PMU_PIDR2.REVISION as an extension to the Part number must use this field as a major revision number.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer modified. Indicates someone other than the Designer has modified the component.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.211 PMU_PIDR4, Performance Monitoring Unit Peripheral Identification Register 4

The PMU_PIDR4 characteristics are:

**Purpose**
Provides information to identify a Performance Monitoring Unit component.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

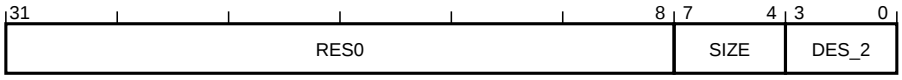
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003FD0`.

This register is not banked between Security states.

## Field descriptions

The PMU_PIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
Size of the component. RAZ. $Log_2$ of the number of 4KB pages from the start of the component to the end of the component ID registers.

**DES_2, bits [3:0]**
Designer, JEP106 continuation code, least significant nibble. For Arm Limited, this field is `0b0100`.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.212 PMU_SWINC, Performance Monitoring Unit Software Increment Register

The PMU_SWINC characteristics are:

**Purpose**
Increments a counter that is configured to count the Software increment event, event `0x00`.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

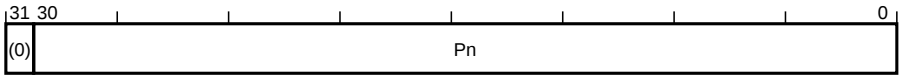
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit write-only register located at `0xE0003CA0`.

This register is not banked between Security states.

## Field descriptions

The PMU_SWINC bit assignments are:




**Bit [31]**
Reserved, RES0.

**Pn, bits [30:0]**
Event counter software increment bit for PMU_EVCNTR<n>. An event counter n, configured for SW_INCR events, increments on every write to bit n of this field.

The possible values of this field are:

**`0b0`**
No action. The write to this bit is ignored.

**`0b1`**
A SW_INCR event is generated for event counter n.

**Note**

Bits [30:N] are WI, where N is the number of counters and the value of PMU_TYPE.N.

This field reads as zero.

### D1.2.213 PMU_TYPE, Performance Monitoring Unit Type Register

The PMU_TYPE characteristics are:

**Purpose**
Contains information regarding what the Performance Monitoring Unit supports.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the PMU is implemented.

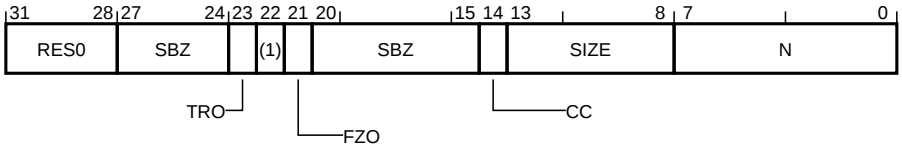
This register is RES0 if the PMU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0003E00`.

This register is not banked between Security states.

## Field descriptions

The PMU_TYPE bit assignments are:




**Bits [31:28]**
Reserved, RES0.

**Bits [27:24]**
SBZ.

**TRO, bit [23]**
Trace-on-overflow support. Identifies whether the trace-on-overflow function is supported.

The possible values of this bit are:

**`0b0`**
Trace-on-overflow not supported.

**`0b1`**
Trace-on-overflow supported.

This bit reads as one.

**Bit [22]**
SBZ.

**FZO, bit [21]**
Freeze-on-overflow support. Identifies whether the freeze-on-overflow mechanism is supported.

The possible values of this bit are:

**0b0**
Freeze-on-overflow mechanism not supported.

**0b1**
Freeze-on-overflow mechanism supported.

This bit reads as one.

**Bits [20:15]**
SBZ.

**CC, bit [14]**
Cycle counter present. This bit is set if a dedicated cycle counter is present.

This bit reads as one.

**SIZE, bits [13:8]**
Size of counters. This field determines the spacing of counters in the memory-map.

**Note**

In ARMv8-M this indicates all counters are word-aligned, as the largest counter is PMU_CCNTR with 32-bits.

This field reads as `0b011111`.

**N, bits [7:0]**
Number of counters.

Number of counters implemented in addition to the cycle counter, PMU_CCNTR.

00000010 PMU_CCNTR and 2 event counters implemented.

00000011 PMU_CCNTR and 3 event counters implemented.

and so on up to 00011111, which indicates PMU_CCNTR and 31 event counters implemented.

**Note**

This field will be non-zero when the PMU is implemented, and serves to indicate the PMU is supported. The maximum number of event counters is 31, so bits[7:5] are always zero.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.214 PRIMASK, Exception Mask Register

The PRIMASK characteristics are:

**Purpose**
Provides access to the PE PRIMASK register.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

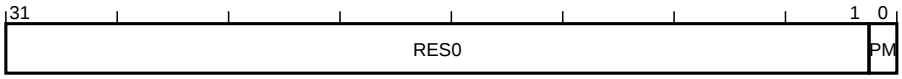
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The PRIMASK bit assignments are:




**Bits [31:1]**
Reserved, RES0.

**PM, bit [0]**
Exception mask register. Setting the Secure PRIMASK to one raises the execution priority to 0. Setting the Non-secure PRIMASK to one raises the execution priority to 0 if AIRCR.PRIS is clear, or `0x80` if AIRCR.PRIS is set.

The possible values of this bit are:

**0**
No effect on execution priority.

**1**
Boosts execution priority to either 0 or `0x80`.

This bit resets to zero on a Warm reset.

### D1.2.215 PSPLIM, Process Stack Pointer Limit Register

The PSPLIM characteristics are:

**Purpose**
Holds the lower limit for the Process stack pointer.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

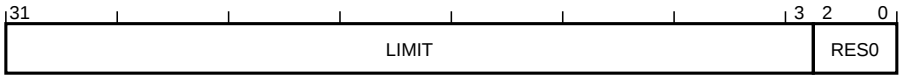
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is banked between Security states.

## Field descriptions

The PSPLIM bit assignments are:




**LIMIT, bits [31:3]**
Stack limit. Bits [31:3] of the Process stack limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure PSPLIM is RES0.

This field resets to zero on a Warm reset.

**Bits [2:0]**
Reserved, RES0.

## D1.2.216 Rn, General-Purpose Register, n = 0 - 12

The R{0..12} characteristics are:

**Purpose**
General-purpose register.

**Usage constraints**
Both privileged and unprivileged accesses are permitted.

This register is word, halfword, and byte accessible.

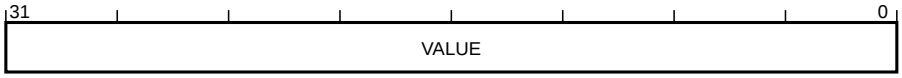
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register.

This register is not banked between Security states.

### Field descriptions

The R{0..12} bit assignments are:




**VALUE, bits [31:0]**
General purpose register value. Armv8-M implemented thirteen general-purpose 32-bit registers, R0 to R12.

This field resets to an UNKNOWN value on a Warm reset.

## D1.2.217 RETPSR, Combined Exception Return Program Status Registers

The RETPSR characteristics are:

**Purpose**
Value pushed to the stack on exception entry. On exception return this is used to restore the flags and other architectural state. This payload is also used for FNC_RETURN stacking, however in this case only some of the fields are used. See FunctionReturn() for details.

**Usage constraints**
None.

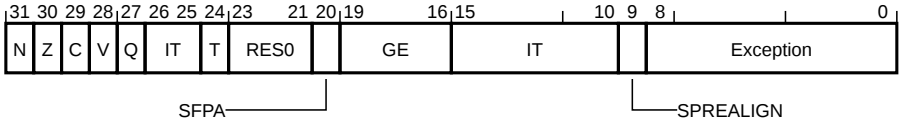
**Configurations**
All.

**Attributes**
32-bit payload.

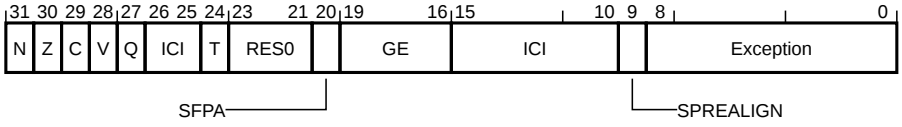
### Field descriptions

The RETPSR bit assignments are:

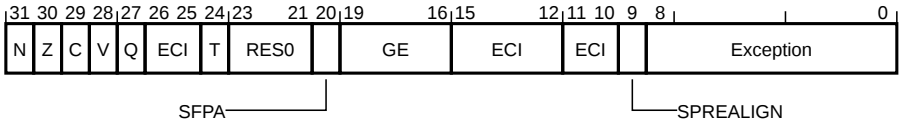
**When {RETPSR[26:25], RETPSR[11:10]} != 0**:




**When {RETPSR[26:25], RETPSR[11:10]} == 0, and a multi-cycle load or store instruction was in progress when the exception was taken**:




**When {RETPSR[26:25], RETPSR[11:10]} == 0, and beat-wise vector instructions were in progress when the exception was taken**:




**N, bit [31]**
Negative flag. Value corresponding to APSR.N.

**Z, bit [30]**
Zero flag. Value corresponding to APSR.Z.

**C, bit [29]**
Carry flag. Value corresponding to APSR.C.

**V, bit [28]**
Overflow flag. Value corresponding to APSR.V.

**Q, bit [27]**
Saturate flag. Value corresponding to APSR.Q.

**T, bit [24]**
T32 state. Value corresponding to EPSR.T.

**Bits [23:21]**
Reserved, RES0.

**SFPA, bit [20]**
Secure Floating-point active. Value corresponding to CONTROL.SFPA.

**GE, bits [19:16]**
Greater-than or equal flag. Value corresponding to APSR.GE.

**IT, bits [15:10,26:25]** , when [{RETPSR[26:25], RETPSR[11:10]} != 0]
If-then flags. Value corresponding to EPSR.IT.

**ICI, bits [26:25,15:10]** , when [{RETPSR[26:25], RETPSR[11:10]} == 0, and a multi-cycle load or store instruction was in progress when the exception was taken]
Interrupt continuation flags. Value corresponding to EPSR.ICI.

**ECI, bits [26:25, 11:10, 15:12]** , when [{RETPSR[26:25], RETPSR[11:10]} == 0, and beat-wise vector instructions were in progress when the exception was taken]
Exception continuation flags for beat-wise vector instructions. Value corresponding to EPSR.ECI.

**SPREALIGN, bit [9]**
Stack-pointer re-align. Indicates whether the SP was re-aligned to an 8-byte alignment on exception entry.

The possible values of this bit are:

**0**
The stack pointer was 8-byte aligned before exception entry began, no special handling is required on exception return.

**1**
The stack pointer was only 4-byte aligned before exception entry. The exception entry realigned SP to 8-byte alignment by increasing the stack frame size by 4-bytes.

**Exception, bits [8:0]**
Exception number. Value corresponding to IPSR.Exception.

### D1.2.218 REVIDR, Revision ID Register

The REVIDR characteristics are:

**Purpose**
Provides implementation-specific minor revision information.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if Armv8.1-M is implemented.

This register is RES0 if Armv8.1-M is not implemented.

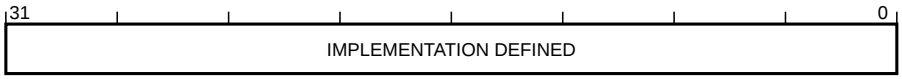
**Attributes**
32-bit read-only register located at `0xE000ECFC`.

Secure software can access the Non-secure version of this register via REVIDR_NS located at `0xE002ECFC`. The location `0xE002ECFC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The REVIDR bit assignments are:




**IMPLEMENTATION DEFINED, bits [31:0]**
IMPLEMENTATION DEFINED. The contents of this field are IMPLEMENTATION DEFINED.

## D1.2.219 RFSR, RAS Fault Status Register

The RFSR characteristics are:

**Purpose**
Records syndrome information for a RAS exception.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if RAS is implemented.

This register is RES0 if RAS is not implemented.

**Attributes**
32-bit read/write register located at `0xE000EF04`.

Secure software can access the Non-secure version of this register via RFSR_NS located at `0xE002EF04`. The location `0xE002EF04` is RES0 to software executing in Non-secure state and the debugger.

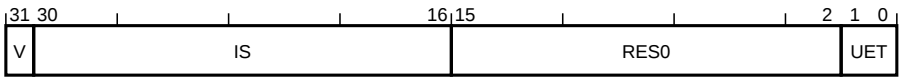
This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

If AIRCR.BFHFNMINS is zero this register is RAZ/WI from Non-secure state.

### Field descriptions

The RFSR bit assignments are:




**V, bit [31]**
Valid. Indicates the register values are valid.

This bit is write-one-to-clear. Writes of zero are ignored.

This bit resets to zero on a Warm reset.

**IS, bits [30:16]**
IMPLEMENTATION DEFINED Syndrome. Contains additional IMPLEMENTATION DEFINED syndrome information.

**Bits [15:2]**
Reserved, RES0.

**UET, bits [1:0]**
Error Type. Describes the state of the processor after taking the RAS exception.

The possible values of this field are:

**`0b00`**
Uncontainable error (UC).

**`0b01`**
Unrecoverable error (UEU).

**0b10**
Restartable error (UEO).

**0b11**
Recoverable error (UER).

This field resets to zero on a Warm reset.

## D1.2.220 SAU_CTRL, SAU Control Register

The SAU_CTRL characteristics are:

**Purpose**
Allows enabling of the Security Attribution Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDD0`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

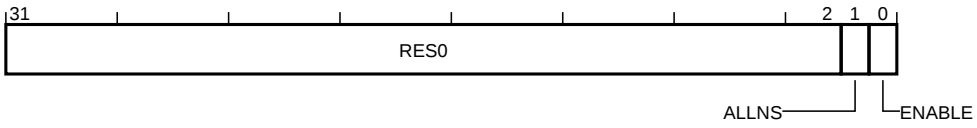
### Preface

It is IMPLEMENTATION DEFINED whether this register:

- Resets to `0x0` - in this case SAU_REGIONn registers are UNKNOWN at reset.
- Resets to an IMPLEMENTATION DEFINED value.

### Field descriptions

The SAU_CTRL bit assignments are:




**Bits [31:2]**
Reserved, RES0.

**ALLNS, bit [1]**
All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure.

The possible values of this bit are:

**0**
Memory is marked as Secure and is not Non-secure callable.

**1**
Memory is marked as Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**ENABLE, bit [0]**

Enable. Enables the SAU.

The possible values of this bit are:

**0**

The SAU is disabled.

**1**

The SAU is enabled.

If this register resets to 1, the SAU region registers also reset to an IMPLEMENTATION DEFINED value.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

### D1.2.221 SAU_RBAR, SAU Region Base Address Register

The SAU_RBAR characteristics are:

**Purpose**
Provides indirect read and write access to the base address of the currently selected SAU region.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDDC`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

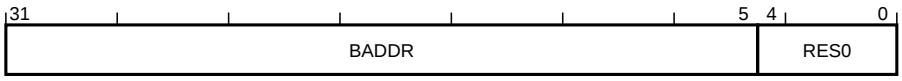
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The SAU_RBAR bit assignments are:




**BADDR, bits [31:5]**
Base address. Holds bits [31:5] of the base address for the selected SAU region.

Bits [4:0] of the base address are defined as `0x00`.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**Bits [4:0]**
Reserved, RES0.

### D1.2.222 SAU_RLAR, SAU Region Limit Address Register

The SAU_RLAR characteristics are:

**Purpose**
Provides indirect read and write access to the limit address of the currently selected SAU region.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDE0`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

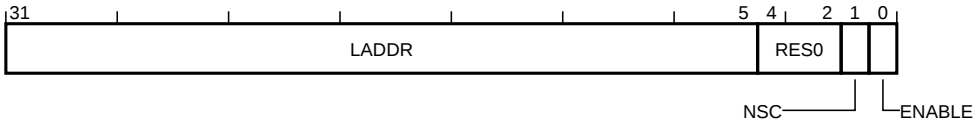
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The SAU_RLAR bit assignments are:




**LADDR, bits [31:5]**
Limit address. Holds bits [31:5] of the limit address for the selected SAU region.

Bits [4:0] of the limit address are defined as `0x1F`.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**Bits [4:2]**
Reserved, RES0.

**NSC, bit [1]**
Non-secure callable. Controls whether Non-secure state is permitted to execute an SG instruction from this region.

The possible values of this bit are:

**0**
Region is not Non-secure callable.

**1**
Region is Non-secure callable.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**ENABLE, bit [0]**
Enable. SAU region enable.

The possible values of this bit are:

**0**
SAU region is disabled.

**1**
SAU region is enabled.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

### D1.2.223 SAU_RNR, SAU Region Number Register

The SAU_RNR characteristics are:

**Purpose**
Selects the region currently accessed by SAU_RBAR and SAU_RLAR.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000EDD8`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

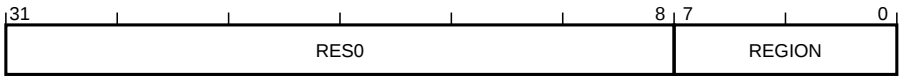
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The SAU_RNR bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REGION, bits [7:0]**
Region number. Indicates the SAU region accessed by SAU_RBAR and SAU_RLAR.

If no SAU regions are implemented, this field is RES0. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.224 SAU_TYPE, SAU Type Register

The SAU_TYPE characteristics are:

**Purpose**
Indicates the number of regions implemented by the Security Attribution Unit.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

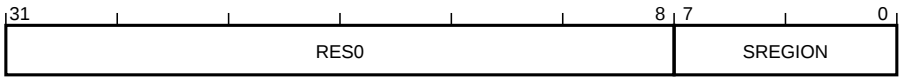
**Attributes**
32-bit read-only register located at `0xE000EDD4`.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

## Field descriptions

The SAU_TYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SREGION, bits [7:0]**
SAU regions. The number of implemented SAU regions.

If this field is RAZ, the SAU behaves as follows:

- SAU_CTRL.ENABLE behaves as RAZ/WI.
- It is IMPLEMENTATION DEFINED whether SAU_CTRL.ALLNS behaves as RAO/WI and all attribution is performed by the IDAU.
- SAU_RNR, SAU_RBAR, and SAU_RLAR behave as RAZ/WI.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.225 SCR, System Control Register

The SCR characteristics are:

**Purpose**
Sets or returns system control data.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED10`.

Secure software can access the Non-secure version of this register via SCR_NS located at `0xE002ED10`. The location `0xE002ED10` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The SCR bit assignments are:


**Bits [31:5]**
Reserved, RES0.

**SEVONPEND, bit [4]**
Send event on pend. Determines whether an interrupt assigned to the same Security state as the SEVONPEND bit transitioning from inactive state to pending state generates a wakeup event.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Transitions from inactive to pending are not wakeup events.

**1**
Transitions from inactive to pending are wakeup events.

This bit resets to zero on a Warm reset.

**SLEEPDEEPS, bit [3]**
Sleep deep secure. This field controls whether the SLEEPDEEP bit is only accessible from the Secure state.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
The SLEEPDEEP bit accessible from both Security states.

**1**
The SLEEPDEEP bit behaves as RAZ/WI when accessed from the Non-secure state.

This bit is only accessible from the Secure state, and behaves as RAZ/WI when accessed from the Non-secure state. If a PE does not implement the deep sleep state this bit behaves as RAZ/WI from both Security states.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**SLEEPDEEP, bit [2]**
Sleep deep. Provides a qualifying hint indicating that waking from sleep might take longer. An implementation can use this bit to select between two alternative sleep states.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
Selected sleep state is not deep sleep.

**1**
Selected sleep state is deep sleep.

Details of the implemented sleep states, if any, and details of the use of this bit, are IMPLEMENTATION DEFINED. If the PE does not implement a deep sleep state then this bit can be RAZ/WI.

This bit resets to zero on a Warm reset.

**SLEEPONEXIT, bit [1]**
Sleep on exit. Determines whether, on an exit from an ISR that returns to the base level of execution priority, the PE enters a sleep state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
Enter sleep state disabled.

**1**
Enter sleep state permitted.

The Secure version of this field is used if the Background state being returned to is the Secure state, otherwise the Non-secure version is used.

This bit resets to zero on a Warm reset.

**Bit [0]**
Reserved, RES0.

### D1.2.226 SFAR, Secure Fault Address Register

The SFAR characteristics are:

**Purpose**
Shows the address of the memory location that caused a Security violation.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000EDE8`.

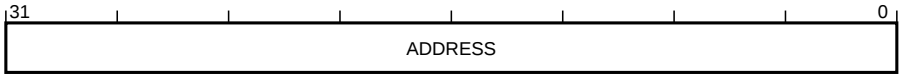
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The SFAR bit assignments are:




**ADDRESS, bits [31:0]**
Address. The address of an access that caused an attribution unit violation. This field is only valid when SFSR.SFARVALID is set. This allows the actual flip flops associated with this register to be shared with other fault address registers. If an implementation chooses to share the storage in this way, care must be taken to not leak Secure address information to the Non-secure state. One way of achieving this is to share the SFAR register with the MMFAR_S register, which is not accessible to the Non-secure state.

This field resets to an UNKNOWN value on a Warm reset.

## D1.2.227 SFSR, Secure Fault Status Register

The SFSR characteristics are:

**Purpose**
Provides information about any security related faults.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write-one-to-clear register located at `0xE000EDE4`.

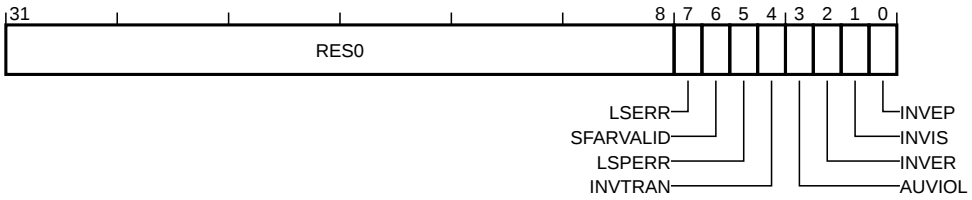
This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The SFSR bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**LSERR, bit [7]**
Lazy state error flag. Sticky flag indicating that an error occurred during lazy state activation or deactivation.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**SFARVALID, bit [6]**
Secure fault address valid. This bit is set when the SFAR register contains a valid value. As with similar fields, such as BFSR.BFARVALID and MMFSR.MMARVALID, this bit can be cleared by other exceptions, such as BusFault.

The possible values of this bit are:

**0**
SFAR content not valid.

**1**
SFAR content valid.

This bit resets to zero on a Warm reset.

**LSPERR, bit [5]**
Lazy state preservation error flag. Stick flag indicating that an SAU or IDAU violation occurred during the lazy preservation of Floating-point state.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVTRAN, bit [4]**
Invalid transition flag. Sticky flag indicating that an exception was raised due to a branch that was not flagged as being domain crossing causing a transition from Secure to Non-secure memory.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**AUVIOL, bit [3]**
Attribution unit violation flag.

Sticky flag indicating that an attempt was made to access parts of the address space that are marked as Secure with NS-Req for the transaction set to Non-secure.

This bit is not set if the violation occurred during:

- Lazy state preservation, see LSPERR.
- Vector fetches.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVER, bit [2]**
Invalid exception return flag. This can be caused by EXC_RETURN.DCRS being set to 0 when returning from an exception in the Non-secure state, or by EXC_RETURN.ES being set to 1 when returning from an exception in the Non-secure state.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVIS, bit [1]**
Invalid integrity signature flag. This bit is set if the integrity signature in an exception stack frame is found to be invalid during the unstacking operation.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVEP, bit [0]**
Invalid entry point. This bit is set if a function call from the Non-secure state or exception targets a non-SG instruction in the Secure state. This bit is also set if the target address is an SG instruction, but there is no matching SAU/IDAU region with the NSC flag set.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

## D1.2.228 SHCSR, System Handler Control and State Register

The SHCSR characteristics are:

**Purpose**
Provides access to the active and pending status of system exceptions.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED24`.

Secure software can access the Non-secure version of this register via SHCSR_NS located at `0xE002ED24`. The location `0xE002ED24` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Preface

Exception processing automatically updates the SHCSR fields. However, software can write to the register to add or remove the pending or active state of an exception. When updating the SHCSR, Arm recommends using a read-modify-write sequence, to avoid unintended effects on the state of the exception handlers.

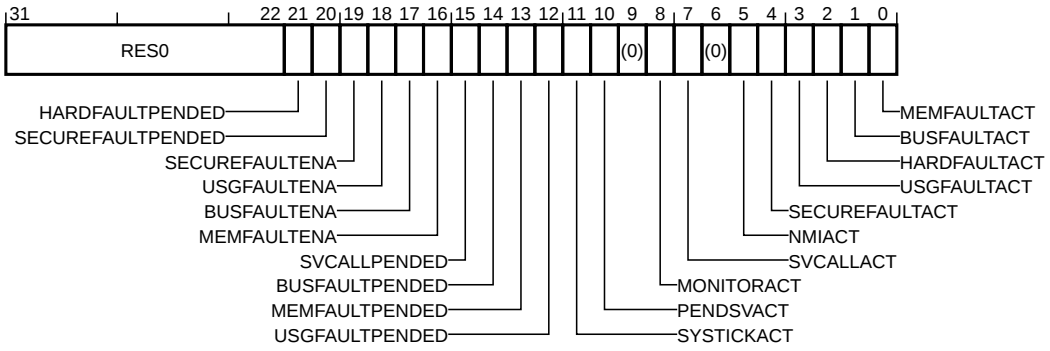
Removing the active state of an exception can change the current execution priority, and affect the exception return consistency checks. If software removes the active state, causing a change in current execution priority, this can defeat the architectural behavior that prevents an exception from preempting its own handler.

Pending state bits are set to one when an exception occurs and are cleared to zero when the exception becomes active.

Active state bits are set to one when the associated exception becomes active.

### Field descriptions

The SHCSR bit assignments are:




**Bits [31:22]**
Reserved, RES0.

**HARDFAULTPENDED, bit [21]**
HardFault exception pended state. This bit indicates and allows modification of the pending state of the HardFault exception corresponding to the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
HardFault exception not pending for the selected Security state.

**1**
HardFault exception pending for the selected Security state.

The Non-secure view of this bit is RAZ/WI if AIRCR.BFHFNMINS is zero.

This bit resets to zero on a Warm reset.

**Note**

The Non-secure HardFault exception will not preempt if AIRCR.BFHFNMINS is set to zero.

**SECUREFAULTPENDED, bit [20]**
SecureFault exception pended state. This bit indicates and allows modification of the pending state of the SecureFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SecureFault exception not pending.

**1**
SecureFault exception pending.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**SECUREFAULTENA, bit [19]**
SecureFault exception enable. The value of this bit defines whether the SecureFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SecureFault exception disabled.

**1**
SecureFault exception enabled.

When disabled, exceptions that target SecureFault escalate to Secure state HardFault.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**USGFAULTENA, bit [18]**
UsageFault exception enable. The value of this bit defines whether the UsageFault exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
UsageFault exception disabled for the selected Security state.

**1**
UsageFault exception enabled for the selected Security state.

When the UsageFault exception is disabled, exceptions targeting UsageFault escalate to HardFault.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**BUSFAULTENA, bit [17]**
BusFault exception enable. The value of this bit defines whether the BusFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
BusFault exception disabled.

**1**
BusFault exception enabled.

The BusFault exception is not banked between Security states. When the BusFault exception is disabled, exceptions targeting BusFault escalate to HardFault.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MEMFAULTENA, bit [16]**
MemManage exception enable. The value of this bit defines whether the MemManage exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
MemManage exception disabled for the selected Security state.

**1**
MemManage exception enabled for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**Note**

When the MemManage exception is disabled, exceptions targeting MemManage escalate to HardFault.

**SVCALLPENDED, bit [15]**
SVCall exception pended state. This bit indicates and allows modification of the pending state of the SVCall exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
SVCall exception not pending for the selected Security state.

**1**
SVCall exception pending for the selected Security state.

This bit resets to zero on a Warm reset.

**BUSFAULTPENDED, bit [14]**
BusFault exception pended state. This bit indicates and allows modification of the pending state of the BusFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
BusFault exception not pending.

**1**
BusFault exception pending.

The BusFault exception is not banked between Security states.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MEMFAULTPENDED, bit [13]**
MemManage exception pended state. This bit indicates and allows modification of the pending state of the MemManage exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
MemManage exception not pending for the selected Security state.

**1**
MemManage exception pending for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**USGFAULTPENDED, bit [12]**
UsageFault exception pended state. The UsageFault exception is banked between Security states, this bit indicates and allows modification of the pending state of the UsageFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
UsageFault exception not pending for the selected Security state.

**1**
UsageFault exception pending for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**SYSTICKACT, bit [11]**
SysTick exception active state. This bit indicates and allows modification of the active state of the SysTick exception for the selected Security state.

If two SysTick timers are implemented this bit is banked between Security states.

If less than two SysTick timers are implemented this bit is not banked between Security states.

The possible values of this bit are:

**0**
SysTick exception not active for the selected Security state.

**1**
SysTick exception active for the selected Security state.

If two timers are implemented, then SYSTICKACT is banked between Security states. If one timer is implemented this bit corresponds to the Secure state if ICSR.STTNS is zero, or the Non-secure state if ICSR.STTNS is one.

This bit resets to zero on a Warm reset.

**PENDSVACT, bit [10]**
PendSV exception active state. This bit indicates and allows modification of the active state of the PendSV exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
PendSV exception not active for the selected Security state.

**1**
PendSV exception active for the selected Security state.

This bit resets to zero on a Warm reset.

**Bit [9]**
Reserved, RES0.

**MONITORACT, bit [8]**
DebugMonitor exception active state. This bit indicates and allows modification of the active state of the DebugMonitor exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
DebugMonitor exception not active.

**1**
DebugMonitor exception active.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**SVCALLACT, bit [7]**
SVCall exception active state. This bit indicates and allows modification of the active state of the SVCall exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
SVCall exception not active for the selected Security state.

**1**
SVCall exception active for the selected Security state.

This bit resets to zero on a Warm reset.

**Bit [6]**
Reserved, RES0.

**NMIACT, bit [5]**
NMI exception active state. This bit indicates and allows modification of the active state of the NMI exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
NMI exception not active.

**1**
NMI exception active.

The NMI exception is not banked between Security states. When AIRCR.BFHFNMINS is zero, the Non-secure view of this bit is RAZ/WI. This field ignores writes if either the value being written is one, AIRCR.BFHFNMINS is zero, the access is from Non-secure state, the access is not via the NS alias, or the access is from a debugger when DHCSR.S_SDE is zero. This bit can only be cleared by access from the Secure state to the NS alias.

This bit resets to zero on a Warm reset.

**SECUREFAULTACT, bit [4]**
SecureFault exception active state. This bit indicates and allows modification of the active state of the SecureFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
SecureFault exception not active.

**1**
SecureFault exception active.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

**USGFAULTACT, bit [3]**
UsageFault exception active state for the selected Security state. This bit indicates and allows modification of the active state of the UsageFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
UsageFault exception not active for the selected Security state.

**1**
UsageFault exception active for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**HARDFAULTACT, bit [2]**
HardFault exception active state. Indicates and allows limited modification of the active state of the HardFault exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
HardFault exception not active for the selected Security state.

**1**
HardFault exception active for the selected Security state.

This field ignores writes if either the value being written is one, the write targets the Secure HardFault active bit, the access is from Non-secure state, or the access is from a debugger when DHCSR.S_SDE is zero.

This bit resets to zero on a Warm reset.

**BUSFAULTACT, bit [1]**
BusFault exception active state. This bit indicates and allows modification of the active state of the BusFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

**0**
BusFault exception not active.

**1**
BusFault exception active.

The BusFault exception is not banked between Security states.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

**MEMFAULTACT, bit [0]**
MemManage exception active state for the selected Security state. This bit indicates and allows modification of the active state of the MemManage exception for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

**0**
MemManage exception not active for the selected Security state.

**1**
MemManage exception active for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

## D1.2.229 SHPR1, System Handler Priority Register 1

The SHPR1 characteristics are:

**Purpose**
Sets or returns priority for system handlers 4 - 7.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
32-bit read/write register located at `0xE000ED18`.

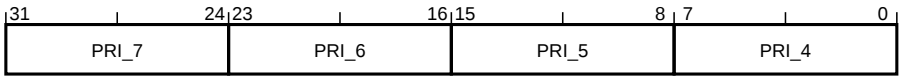
Secure software can access the Non-secure version of this register via SHPR1_NS located at `0xE002ED18`. The location `0xE002ED18` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The SHPR1 bit assignments are:




**PRI_7, bits [31:24]**
Priority 7. Priority of system handler 7, SecureFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

**PRI_6, bits [23:16]**
Priority 6. Priority of system handler 6, UsageFault.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

**PRI_5, bits [15:8]**
Priority 5. Priority of system handler 5, BusFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

**PRI_4, bits [7:0]**

Priority 4. Priority of system handler 4, MemManage.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

## D1.2.230 SHPR2, System Handler Priority Register 2

The SHPR2 characteristics are:

**Purpose**
Sets or returns priority for system handlers 8 - 11.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED1C`.

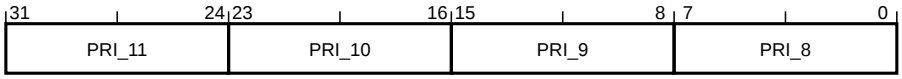
Secure software can access the Non-secure version of this register via SHPR2_NS located at `0xE002ED1C`. The location `0xE002ED1C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The SHPR2 bit assignments are:




**PRI_11, bits [31:24]**
Priority 11. Priority of system handler 11, SVCall.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

**PRI_10, bits [23:16]**
Reserved, RES0.

**PRI_9, bits [15:8]**
Reserved, RES0.

**PRI_8, bits [7:0]**
Reserved, RES0.

## D1.2.231 SHPR3, System Handler Priority Register 3

The SHPR3 characteristics are:

**Purpose**
Sets or returns priority for system handlers 12 - 15.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED20`.

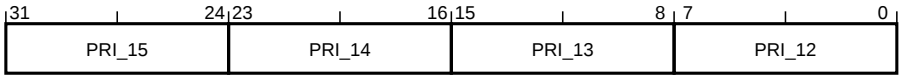
Secure software can access the Non-secure version of this register via SHPR3_NS located at `0xE002ED20`. The location `0xE002ED20` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Field descriptions

The SHPR3 bit assignments are:




**PRI_15, bits [31:24]**
Priority 15. Priority of system handler 15, SysTick.

If two SysTick timers are implemented this field is banked between Security states.

If less than two SysTick timers are implemented this field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0. If one timer is implemented, this field corresponds to the Secure state if ICSR.STTNS is zero, or the Non-secure state if ICSR.STTNS is one.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to zero on a Warm reset.

**PRI_14, bits [23:16]**
Priority 14. Priority of system handler 14, PendSV.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

**PRI_13, bits [15:8]**
Reserved, RES0.

**PRI_12, bits [7:0]**
Priority 12. Priority of system handler 12, DebugMonitor.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

If DEMCR.SDME is one this field is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

## D1.2.232 SP, Current Stack Pointer Register

The SP characteristics are:

**Purpose**
Exception and procedure stack pointer register.

**Usage constraints**
Privileged and unprivileged access permitted.

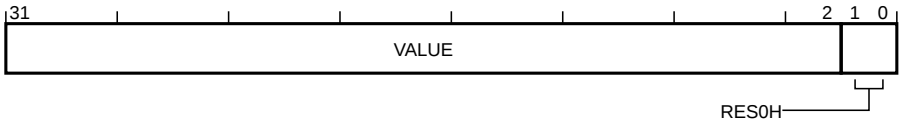
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

### Field descriptions

The SP bit assignments are:




**VALUE, bits [31:2]**
Stack pointer. Holds bits[31:2] of the stack pointer address. The current stack pointer is selected from one of MSP_NS, PSP_NS, MSP_S or PSP_S.

Software can refer to SP as R13.

**Bits [1:0]**
Reserved, RES0H.

### D1.2.233 SP_NS, Stack Pointer (Non-secure)

The SP_NS characteristics are:

**Purpose**
Provides access to the current Non-secure stack pointer.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

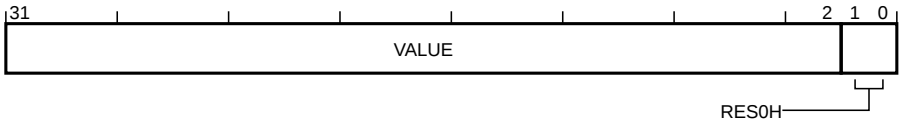
**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The SP_NS bit assignments are:




**VALUE, bits [31:2]**
Stack pointer. Holds bits[31:2] of the current Non-secure stack pointer address. SP_NS is selected from one of MSP_NS or PSP_NS. Access to SP_NS is provided via MRS and MSR and is subject to stack limit checking.

**Bits [1:0]**
Reserved, RES0H.

### D1.2.234 STIR, Software Triggered Interrupt Register

The STIR characteristics are:

**Purpose**
Provides a mechanism for software to generate an interrupt.

**Usage constraints**
Unprivileged accesses generate a fault if CCR.USERSETMPEND is zero.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

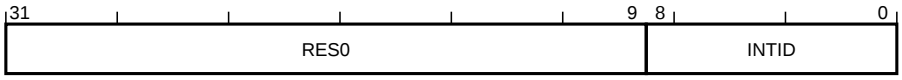
**Attributes**
32-bit write-only register located at `0xE000EF00`.

Secure software can access the Non-secure version of this register via STIR_NS located at `0xE002EF00`. The location `0xE002EF00` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

## Field descriptions

The STIR bit assignments are:




**Bits [31:9]**
Reserved, RES0.

**INTID, bits [8:0], on a write**
Interrupt ID. Indicates the interrupt to be pended. The value written is (ExceptionNumber - 16).

Writing to this register has the same effect as setting the NVIC_ISPR*n* bit corresponding to the interrupt to 1. Like NVIC_ISPR*n*, an attempt to pend an interrupt targeting Secure state from Non-secure is ignored.

**INTID, bits [8:0], on a read**
This field reads as zero.

## D1.2.235 SYST_CALIB, SysTick Calibration Value Register

The SYST_CALIB characteristics are:

**Purpose**
Reads the SysTick timer calibration value and parameters for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if at least one SysTick timer is implemented.

This register is RES0 if no SysTick timer is implemented.

**Attributes**
32-bit read-only register located at `0xE000E01C`.

Secure software can access the Non-secure version of this register via SYST_CALIB_NS located at `0xE002E01C`. The location `0xE002E01C` is RES0 to software executing in Non-secure state and the debugger.

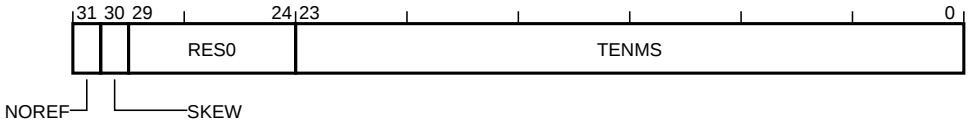
This register is banked between Security states.

### Preface

If the Main Extension is implemented then, two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

### Field descriptions

The SYST_CALIB bit assignments are:




**NOREF, bit [31]**
No reference. Indicates whether the IMPLEMENTATION DEFINED reference clock is implemented.

The possible values of this bit are:

**0**
Reference clock is implemented.

**1**
Reference clock is not implemented.

When this bit is 1, the CLKSOURCE bit of the SYST_CSR register is forced to 1 and cannot be cleared to 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This bit reads as an IMPLEMENTATION DEFINED value.

**SKEW, bit [30]**
Skew. Indicates whether the 10ms calibration value is exact.

The possible values of this bit are:

**0**
TENMS calibration value is exact.

**1**
TENMS calibration value is inexact.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This bit reads as an IMPLEMENTATION DEFINED value.

**Bits [29:24]**
Reserved, RES0.

**TENMS, bits [23:0]**
Ten milliseconds. Optionally holds a reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If this field is zero, the calibration value is not known.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.236 SYST_CSR, SysTick Control and Status Register

The SYST_CSR characteristics are:

**Purpose**
Controls the SysTick timer and provides status data for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if at least one SysTick timer is implemented.

This register is RES0 if no SysTick timer is implemented.

**Attributes**
32-bit read/write register located at `0xE000E010`.

Secure software can access the Non-secure version of this register via SYST_CSR_NS located at `0xE002E010`. The location `0xE002E010` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

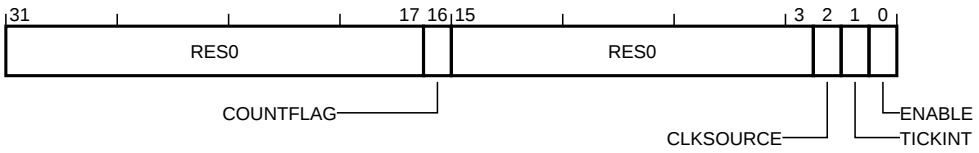
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

## Field descriptions

The SYST_CSR bit assignments are:




**Bits [31:17]**
Reserved, RES0.

**COUNTFLAG, bit [16]**
Count flag. Indicates whether the counter has counted to zero since the last read of this register.

The possible values of this bit are:

**0**
Timer has not counted to 0.

**1**
Timer has counted to 0.

COUNTFLAG is set to 1 by a count transition from 1 to 0. COUNTFLAG is cleared to 0 if software reads this bit as one, and by any write to the SYST_CVR for the selected Security state. Debugger reads do not clear the COUNTFLAG.

If set this bit clears to zero when read by software. Reads from the debugger do not clear this bit.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

**Bits [15:3]**
Reserved, RES0.

**CLKSOURCE, bit [2]**
Clock source. Indicates the SysTick clock source.

The possible values of this bit are:

**0**
Uses the IMPLEMENTATION DEFINED external reference clock.

**1**
Uses the PE clock.

If no external clock is implemented, this bit reads as 1 and ignores writes.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**TICKINT, bit [1]**
Tick interrupt. Indicates whether counting to 0 causes the status of the SysTick exception to change to pending.

The possible values of this bit are:

**0**
Count to 0 does not affect the SysTick exception status.

**1**
Count to 0 changes the SysTick exception status to pending.

Changing the value of the counter to 0 by writing the SysTick does not change the status of the SysTick exception.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

**ENABLE, bit [0]**
SysTick enable. Indicates the enabled status of the SysTick counter.

The possible values of this bit are:

**0**
Counter is disabled.

**1**
Counter is enabled.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

## D1.2.237 SYST_CVR, SysTick Current Value Register

The SYST_CVR characteristics are:

**Purpose**
Reads or clears the SysTick timer current counter value for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if at least one SysTick timer is implemented.

This register is RES0 if no SysTick timer is implemented.

**Attributes**
32-bit read/write-to-clear register located at `0xE000E018`.

Secure software can access the Non-secure version of this register via SYST_CVR_NS located at `0xE002E018`. The location `0xE002E018` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

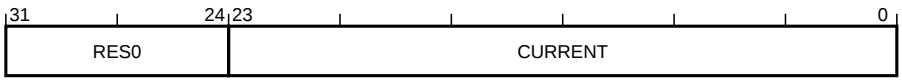
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

### Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

### Field descriptions

The SYST_CVR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**CURRENT, bits [23:0], on a read**
Current counter value. Provides the value of the SysTick timer counter for the selected Security state.

It is IMPLEMENTATION DEFINED whether the current counter value decrements if the PE is sleeping and SCR.SLEEPDEEP is set.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

**CURRENT, bits [23:0], on a write**

Reset counter value. Writing any value clears the SysTick timer counter for the selected Security state to zero.

### D1.2.238 SYST_RVR, SysTick Reload Value Register

The SYST_RVR characteristics are:

**Purpose**
Provides access SysTick timer counter reload value for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if at least one SysTick timer is implemented.

This register is RES0 if no SysTick timer is implemented.

**Attributes**
32-bit read/write register located at `0xE000E014`.

Secure software can access the Non-secure version of this register via SYST_RVR_NS located at `0xE002E014`. The location `0xE002E014` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

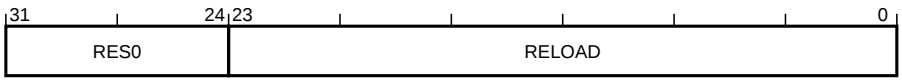
From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both instances of this register behave as RES0.

## Field descriptions

The SYST_RVR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**RELOAD, bits [23:0]**
Counter reload value. The value to load into the SYST_CVR for the selected Security state when the counter reaches 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.239 TPIU_ACPR, TPIU Asynchronous Clock Prescaler Register

The TPIU_ACPR characteristics are:

**Purpose**

Defines a prescaler value for the baud rate of the Serial Wire Output (SWO). Writing to the register automatically updates the prescale counter, immediately affecting the baud rate of the serial data output.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

If a debugger changes the register value while the TPIU is transmitting data, the effect on the output stream is UNPREDICTABLE and the required recovery process is IMPLEMENTATION DEFINED.

**Configurations**

Present only if the TPIU is implemented and supports SWO.

This register is RES0 if the TPIU is not implemented or does not support SWO.

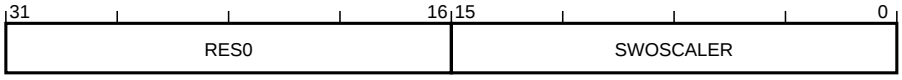
**Attributes**

32-bit read/write register located at `0xE0040010`.

This register is not banked between Security states.

## Field descriptions

The TPIU_ACPR bit assignments are:




**Bits [31:16]**

Reserved, RES0.

**SWOSCALER, bits [15:0]**

SWO baud rate prescalar. Sets the ratio between an IMPLEMENTATION DEFINED reference clock and the SWO output clock rates. The supported scaler value range is IMPLEMENTATION DEFINED, to a maximum scalar value of `0xFFFF`. Unused bits of this field are RAZ/WI.

The possible values of this field are:

***n***

SWO output clock = Asynchronous_Reference_Clock/($n$ + 1).

This field resets to zero on a Cold reset.

## D1.2.240 TPIU_CIDR0, TPIU Component Identification Register 0

The TPIU_CIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

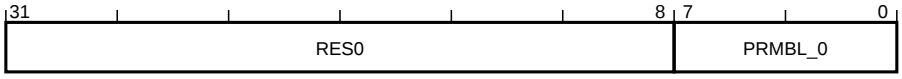
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FF0`.

This register is not banked between Security states.

### Field descriptions

The TPIU_CIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_0, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

### D1.2.241 TPIU_CIDR1, TPIU Component Identification Register 1

The TPIU_CIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

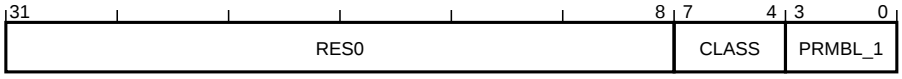
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FF4`.

This register is not banked between Security states.

## Field descriptions

The TPIU_CIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**CLASS, bits [7:4]**
CoreSight component class. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

**PRMBL_1, bits [3:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

## D1.2.242 TPIU_CIDR2, TPIU Component Identification Register 2

The TPIU_CIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

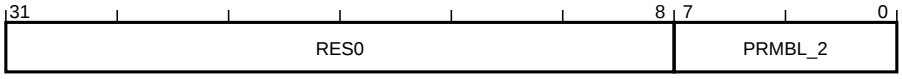
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FF8`.

This register is not banked between Security states.

### Field descriptions

The TPIU_CIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_2, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

### D1.2.243 TPIU_CIDR3, TPIU Component Identification Register 3

The TPIU_CIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

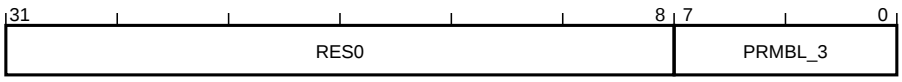
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FFC`.

This register is not banked between Security states.

## Field descriptions

The TPIU_CIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PRMBL_3, bits [7:0]**
CoreSight component identification preamble. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

### D1.2.244 TPIU_CSPSR, TPIU Current Parallel Port Sizes Register

The TPIU_CSPSR characteristics are:

**Purpose**
Controls the width of the parallel trace port.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the TPIU is implemented.

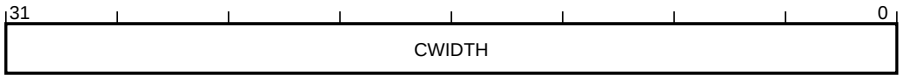
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read/write register located at `0xE0040004`.

This register is not banked between Security states.

## Field descriptions

The TPIU_CSPSR bit assignments are:




**CWIDTH, bits [31:0]**
Current width. CWIDTH[*m*] represents a parallel trace port width of (*m*+1).

The possible values of each bit are:

**0**
Width (N+1) is not the current parallel trace port width.

**1**
Width (N+1) is the current parallel trace port width.

A debugger must set only one bit is set to 1, and all others must be zero. The effect of writing a value with more than one bit set to 1 is UNPREDICTABLE. The effect of a write to an unsupported bit is UNPREDICTABLE.

This register resets to the value for the smallest supported parallel trace port size.

This field resets to an IMPLEMENTATION DEFINED value on a Cold reset.

### D1.2.245 TPIU_DEVTYPE, TPIU Device Type Register

The TPIU_DEVTYPE characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

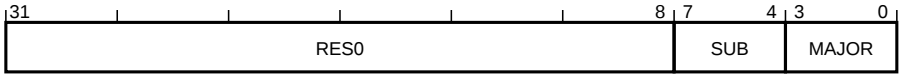
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FCC`.

This register is not banked between Security states.

## Field descriptions

The TPIU_DEVTYPE bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SUB, bits [7:4]**
Sub-type. Component sub-type.

The possible values of this field are:

**0x0**
Other. Only permitted if the MAJOR field reads as `0x0`.

**0x1**
Trace port. Only permitted if the MAJOR field reads as `0x1`.

This field reads as an IMPLEMENTATION DEFINED value.

**MAJOR, bits [3:0]**
Major type. Component major type.

The possible values of this field are:

**0x0**
Miscellaneous.

**0x1**
Trace sink.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.246 TPIU_FFCR, TPIU Formatter and Flush Control Register

The TPIU_FFCR characteristics are:

**Purpose**

Controls the TPIU formatter. This register might contain other formatter and flush control fields that are outside the scope of the architecture. Contact Arm for more information.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

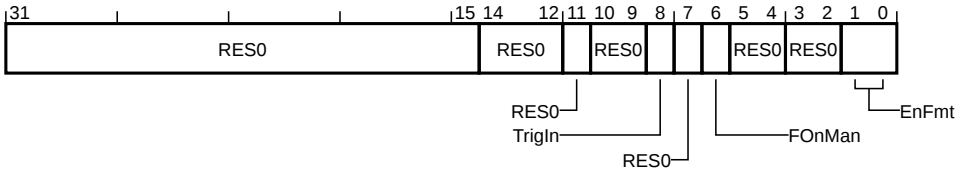
**Attributes**

32-bit read/write register located at `0xE0040304`.

This register is not banked between Security states.

## Field descriptions

The TPIU_FFCR bit assignments are:




**Bits [31:15,11,7,3:2]**

Reserved, RES0.

**Bits [14:12]**

Reserved for formatter stop controls.

Reserved, RES0.

**Bits [10:9]**

Reserved for additional trigger mark controls.

Reserved, RES0.

**TrigIn, bit [8]**

Trigger input asserted. Indicate a trigger on the trace port when an IMPLEMENTATION DEFINED TRIGIN signal is asserted.

It is IMPLEMENTATION DEFINED whether this bit is R/W or RAO.

This bit resets to zero on a Cold reset.

**FOnMan, bit [6]**
Flush On Manual. Setting this bit to 1 generates a flush. The TPIU clears the bit to 0 when the flush completes.

This bit resets to zero on a Cold reset.

**Bits [5:4]**
Reserved for additional flush controls.

Reserved, RES0.

**EnFmt, bits [1:0]**
Formatter control. Selects the output formatting mode.

The possible values of this field are:

**0b00**
Bypass. Disable formatting. Only supported when SWO mode is selected. Only a single trace source is supported in bypass mode:

- If only a single trace source is connected to this TPIU, it is selected.
- If multiple sources (including the ITM) are implemented and connected to this TPIU, then all other trace sources, except for the ITM, must be disabled. Otherwise, the trace output is UNPREDICTABLE.

All other trace sources are discarded.

**0b10**
Continuous. Enable formatting and embed triggers and null cycles in the formatted output.

All other values are reserved.

If no formatter is implemented, this field is RES0. This field must be set to `0b10` when the parallel trace port is selected, or when using multiple trace sources. Changing the value of this field when TPIU_FFSR.FtStopped is 0 is UNPREDICTABLE.

This field resets to zero on a Cold reset.

**Note**

An optional TRACECTL pin might be implemented as part of the parallel trace port that allows Bypass mode when using a parallel trace port and a further mode, EnFmt == `0b01`. The CoreSight architecture describes EnFmt[1] as the EnFCont bit and EnFmt[0] as the EnFTC bit.

### D1.2.247 TPIU_FFSR, TPIU Formatter and Flush Status Register

The TPIU_FFSR characteristics are:

**Purpose**
Shows the status and capabilities of the TPIU formatter.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the TPIU is implemented.

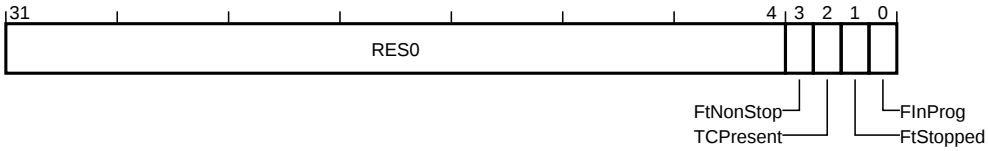
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040300`.

This register is not banked between Security states.

## Field descriptions

The TPIU_FFSR bit assignments are:




**Bits [31:4]**
Reserved, RES0.

**FtNonStop, bit [3]**
Non-stop formatter. Indicates the formatter cannot be stopped.

The possible values of this bit are:

**0**
Formatter can be stopped.

**1**
Formatter cannot be stopped.

If no formatter is implemented, this bit is RAO.

**TCPresent, bit [2]**
TRACECTL present. Indicates presence of the TRACECTL pin.

The possible values of this bit are:

**0**
No TRACECTL pin is available. The data formatter must be used and only in continuous mode.

**1**
The optional TRACECTL pin is present.

If a parallel trace port is not implemented, this bit is RAZ.

**Note**

If a parallel trace port is implemented, Arm recommends the TRACECTL pin is not implemented.

**FtStopped, bit [1]**
Formatter stopped. Indicates the formatter is stopped.

The possible values of this bit are:

**0**
Formatter is enabled.

**1**
The formatter has received a stop request signal and all trace data and post-amble has been output. Any further trace data is ignored.

If no formatter is implemented, or the formatter cannot be stopped, this bit is RAZ.

**FInProg, bit [0]**
Flush in progress. Set to 1 when a flush is initiated and clears to zero when all data received before the flush is acknowledged has been output on the trace port. That is, the trace has been received at the sink, formatted, and output on the trace port.

### D1.2.248 TPIU_LAR, TPIU Software Lock Access Register

The TPIU_LAR characteristics are:

**Purpose**

Provides CoreSight Software Lock control for the TPIU, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

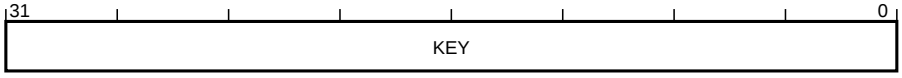
**Attributes**

32-bit write-only register located at `0xE0040FB0`.

This register is not banked between Security states.

## Field descriptions

The TPIU_LAR bit assignments are:




**KEY, bits [31:0]**

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to the registers of this component through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to the registers of this component through a memory mapped interface.

### D1.2.249 TPIU_LSR, TPIU Software Lock Status Register

The TPIU_LSR characteristics are:

**Purpose**

Provides CoreSight Software Lock status information for the TPIU, see the *Arm® CoreSight™ Architecture Specification* for details.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

This register is RAZ/WI if accessed via the debugger.

**Configurations**

Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

Present only if the optional Software Lock is implemented.

This register is RAZ/WI if the Software Lock is not implemented.

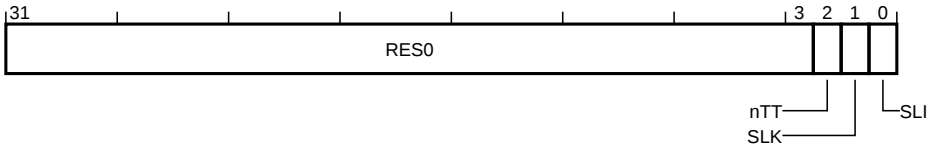
**Attributes**

32-bit read-only register located at `0xE0040FB4`.

This register is not banked between Security states.

## Field descriptions

The TPIU_LSR bit assignments are:




**Bits [31:3]**

Reserved, RES0.

**nTT, bit [2]**

Not thirty-two bit. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as zero.

**SLK, bit [1]**

Software Lock status. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Lock clear. Software writes are permitted to the registers of this component.

**1**

Lock set. Software writes to the registers of this component are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Cold reset.

**SLI, bit [0]**

Software Lock implemented. See the *Arm® CoreSight™ Architecture Specification*.

The possible values of this bit are:

**0**

Software Lock not implemented or debugger access.

**1**

Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

## D1.2.250 TPIU_PIDR0, TPIU Peripheral Identification Register 0

The TPIU_PIDR0 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

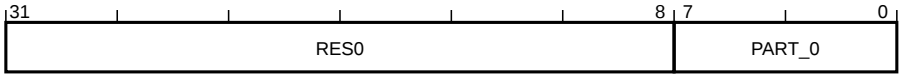
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FE0`.

This register is not banked between Security states.

### Field descriptions

The TPIU_PIDR0 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**PART_0, bits [7:0]**
Part number bits [7:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.251 TPIU_PIDR1, TPIU Peripheral Identification Register 1

The TPIU_PIDR1 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

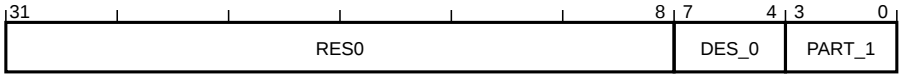
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FE4`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR1 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**DES_0, bits [7:4]**
JEP106 identification code bits [3:0]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**PART_1, bits [3:0]**
Part number bits [11:8]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.252 TPIU_PIDR2, TPIU Peripheral Identification Register 2

The TPIU_PIDR2 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

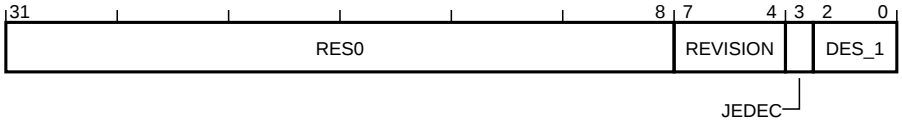
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FE8`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR2 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVISION, bits [7:4]**
Component revision. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**JEDEC, bit [3]**
JEDEC assignee value is used. See the *Arm® CoreSight™ Architecture Specification*.

This bit reads as one.

**DES_1, bits [2:0]**
JEP106 identification code bits [6:4]. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.253 TPIU_PIDR3, TPIU Peripheral Identification Register 3

The TPIU_PIDR3 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

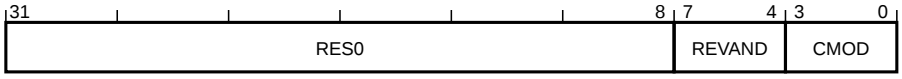
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FEC`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR3 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**REVAND, bits [7:4]**
RevAnd. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

**CMOD, bits [3:0]**
Customer Modified. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.254 TPIU_PIDR4, TPIU Peripheral Identification Register 4

The TPIU_PIDR4 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

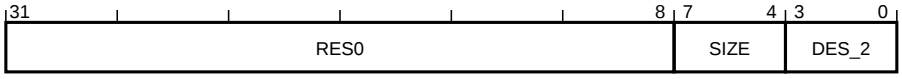
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FD0`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR4 bit assignments are:




**Bits [31:8]**
Reserved, RES0.

**SIZE, bits [7:4]**
4KB count. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as zero.

**DES_2, bits [3:0]**
JEP106 continuation code. See the *Arm® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

### D1.2.255 TPIU_PIDR5, TPIU Peripheral Identification Register 5

The TPIU_PIDR5 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

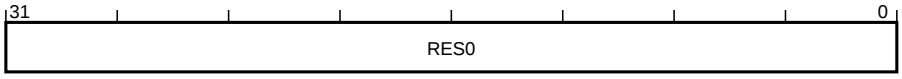
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FD4`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR5 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.256 TPIU_PIDR6, TPIU Peripheral Identification Register 6

The TPIU_PIDR6 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

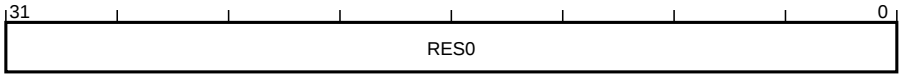
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FD8`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PIDR6 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

## D1.2.257 TPIU_PIDR7, TPIU Peripheral Identification Register 7

The TPIU_PIDR7 characteristics are:

**Purpose**
Provides CoreSight Unique Component Identifier information for the TPIU.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

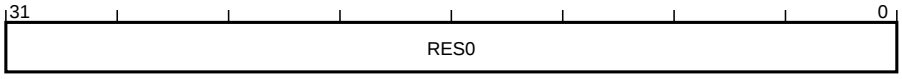
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FDC`.

This register is not banked between Security states.

### Field descriptions

The TPIU_PIDR7 bit assignments are:




**Bits [31:0]**
Reserved, RES0.

### D1.2.258 TPIU_PSCR, TPIU Periodic Synchronization Control Register

The TPIU_PSCR characteristics are:

**Purpose**

Defines the reload value for the Periodic Synchronization Counter register. The Periodic Synchronization Counter decrements for each byte that is output by the TPIU. If the formatter is implemented and enabled, the TPIU forces completion of the current frame when the counter reaches zero. It is IMPLEMENTATION DEFINED whether the TPIU forces all trace sources to generate synchronization packets when the counter reaches zero. Bytes generated by the TPIU as part of a Halfword synchronization packet or a Full frame synchronization packet are not counted.

**Usage constraints**

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**

Present if the TPIU is implemented and DWT_CYCCNT is not implemented.

OPTIONAL if both the TPIU and DWT_CYCCNT are implemented.

This register is RES0 if the TPIU is not implemented.

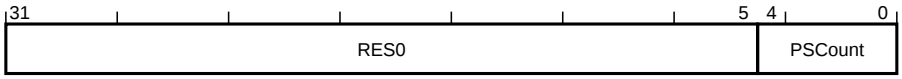
**Attributes**

32-bit read/write register located at `0xE0040308`.

This register is not banked between Security states.

## Field descriptions

The TPIU_PSCR bit assignments are:




**Bits [31:5]**

Reserved, RES0.

**PSCount, bits [4:0]**

Periodic Synchronization Count. Determines the reload value of the Periodic Synchronization Counter. The reload value takes effect the next time the counter reaches zero. Reads from this register return the reload value programmed into this register.

The possible values of this field are:

**`0b00000`**
Synchronization disabled.

**`0b00111`**
128 bytes.

**`0b01000`**
256 bytes.

**...**

...

**0b11111**

$2^{31}$ bytes.

All other values are reserved.

The Periodic Synchronization Counter might have a maximum value smaller than $2^{31}$. In this case, if the programmed reload value is greater than the maximum value, then the Periodic Synchronization Counter is reloaded with its maximum value and the TPIU will generate synchronization requests at this interval.

This field resets to `0xA` on a Cold reset.

**Note**

In the CoreSight TPIU, TPIU_PSCR specifies the number of frames between synchronizations, each frame being 16 bytes. This definition of TPIU_PSCR specifies a number of bytes and is encoded as a power-of-two rather than a plain binary number.

### D1.2.259 TPIU_SPPR, TPIU Selected Pin Protocol Register

The TPIU_SPPR characteristics are:

**Purpose**
Selects the protocol used for trace output.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

If a debugger changes the register value while the TPIU is transmitting data, the effect on the output stream is UNPREDICTABLE and the required recovery process is IMPLEMENTATION DEFINED.

**Configurations**
Present only if the TPIU is implemented and supports SWO.

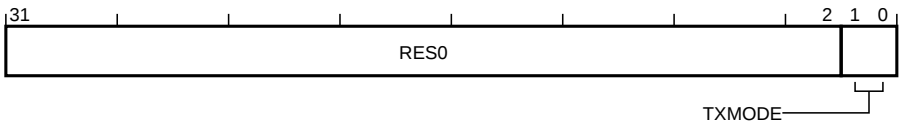
This register is RES0 if the TPIU is not implemented or does not support SWO.

**Attributes**
32-bit read/write register located at `0xE00400F0`.

This register is not banked between Security states.

## Field descriptions

The TPIU_SPPR bit assignments are:




**Bits [31:2]**
Reserved, RES0.

**TXMODE, bits [1:0]**
Transmit mode. Specifies the protocol for trace output from the TPIU.

The possible values of this field are:

**`0b00`**
Parallel trace port mode. This value is reserved if TPIU_TYPE.PTINVALID == 1.

**`0b01`**
Asynchronous SWO, using Manchester encoding. This value is reserved if TPIU_TYPE.MANCVALID == 0.

**`0b10`**
Asynchronous SWO, using NRZ encoding. This value is reserved if TPIU_TYPE.NRZVALID == 0.

All other values are reserved.

The effect of selecting a reserved value, or a mode that the implementation does not support, is UNPREDICTABLE.

This field resets to an IMPLEMENTATION DEFINED value on a Cold reset.

## D1.2.260 TPIU_SSPSR, TPIU Supported Parallel Port Sizes Register

The TPIU_SSPSR characteristics are:

**Purpose**
Indicates the supported parallel trace port sizes.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the TPIU is implemented.

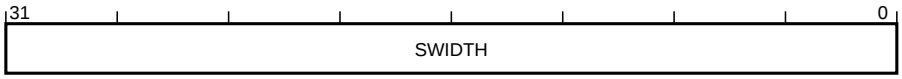
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040000`.

This register is not banked between Security states.

### Field descriptions

The TPIU_SSPSR bit assignments are:




**SWIDTH, bits [31:0]**
Supported width. SWIDTH[*m*] indicates whether a parallel trace port width of (*m*+1) is supported.

The possible values of each bit are:

**0**
Parallel trace port width (*m*+1) not supported.

**1**
Parallel trace port width (*m*+1) supported.

The value of this register is IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value.

## D1.2.261 TPIU_TYPE, TPIU Device Identifier Register

The TPIU_TYPE characteristics are:

**Purpose**
Describes the TPIU to a debugger.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

**Configurations**
Present only if the TPIU is implemented.

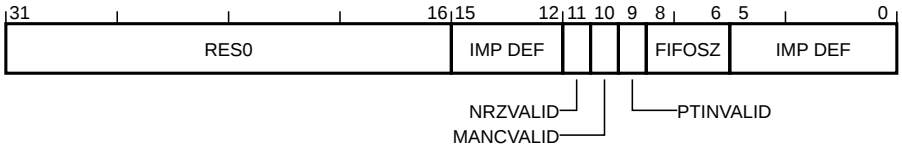
This register is RES0 if the TPIU is not implemented.

**Attributes**
32-bit read-only register located at `0xE0040FC8`.

This register is not banked between Security states.

### Field descriptions

The TPIU_TYPE bit assignments are:




**Bits [31:16]**
Reserved, RES0.

**Bits [15:12]**
IMPLEMENTATION DEFINED.

**NRZVALID, bit [11]**
NRZ valid. Indicates support for SWO using UART/NRZ encoding.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**MANCVALID, bit [10]**
Manchester valid. Indicates support for SWO using Manchester encoding.

The possible values of this bit are:

**0**
Not supported.

**1**
Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**PTINVALID, bit [9]**
Parallel Trace Interface invalid. Indicates support for parallel trace port operation.

The possible values of this bit are:

**0**
Supported.

**1**
Not supported.

This bit reads as an IMPLEMENTATION DEFINED value.

**FIFOSZ, bits [8:6]**
FIFO depth. Indicates the minimum implemented size of the TPIU output FIFO for trace data.

The possible values of this field are:

**0**
IMPLEMENTATION DEFINED FIFO depth.

**Other**
Minimum FIFO size is $2^{FIFOSZ}$.

For example, a value of `0b011` indicates a FIFO size of at least $2^3$ = 8 bytes.

This field reads as an IMPLEMENTATION DEFINED value.

**Bits [5:0]**
IMPLEMENTATION DEFINED.

## D1.2.262 TT_RESP, Test Target Response Payload

The TT_RESP characteristics are:

**Purpose**
Provides the response information from a TT, TTA, TTT, or TTAT instruction.

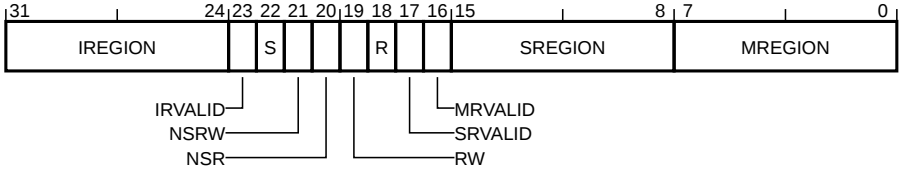
**Usage constraints**
None.

**Configurations**
All.

**Attributes**
32-bit payload.

### Field descriptions

The TT_RESP bit assignments are:




**IREGION, bits [31:24]**
IDAU region number. Indicates the IDAU region number containing the target address.

This field is zero if IRVALID is zero.

**IRVALID, bit [23]**
IREGION valid flag. For a Secure request, indicates the validity of the IREGION field.

The possible values of this bit are:

**0**
IREGION content not valid.

**1**
IREGION content valid.

This bit is always zero if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction was executed from the Non-secure state.

**S, bit [22]**
Security. For a Secure request, indicates the Security attribute of the target address.

The possible values of this bit are:

**0**
Target address is Non-secure.

**1**
Target address is Secure.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

**NSRW, bit [21]**
Non-secure read and writable. Equal to RW AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the RW field is valid.

**NSR, bit [20]**
Non-secure readable. Equal to R AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the R field is valid.

**RW, bit [19]**
Read and writable.

Set to 1 if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

**R, bit [18]**
Readable.

Read accessibility. Set to 1 if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

**SRVALID, bit [17]**
SREGION valid flag. For a Secure request indicates validity of the SREGION field.

The possible values of this bit are:

**0**
SREGION content not valid.

**1**
SREGION content valid.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

The SREGION field is invalid if any of the following are true:

- SAU_CTRL.ENABLE is set to zero.
- The register argument specified in the SREGION field does not match any enabled SAU regions.
- The address specified matches multiple enabled SAU regions.
- The address specified by the SREGION field is exempt from the secure memory attribution.
- The TT instruction was executed from the Non-secure state or the Security Extension is not implemented.

**MRVALID, bit [16]**
MREGION valid flag. Indicates validity of the MREGION field.

The possible values of this bit are:

**0**
MREGION content not valid.

**1**
MREGION content valid.

This bit is only valid for TT and TTA instructions, executed in the Secure state or in privileged mode in Non-secure state.

The MREGION field is invalid if any of the following is true:

- The MPU is not implemented or MPU_CTRL.ENABLE is set to zero.

- The register argument specified by the MREGION field does not match any enabled MPU regions.
- The address matched multiple MPU regions.
- The TT instruction was executed from an unprivileged mode and the A flag was not specified.

**SREGION, bits [15:8]**
SAU region number. Holds the SAU region that the address maps to.

This field is only valid if the instruction was executed from Secure state. This field is zero if SRVALID is 0.

**MREGION, bits [7:0]**
MPU region number. Holds the MPU region that the address maps to.

This field is zero if MRVALID is 0.

### D1.2.263 UFSR, UsageFault Status Register

The UFSR characteristics are:

**Purpose**
Contains the status for some instruction execution faults, and for data access faults.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

**Configurations**
Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

**Attributes**
16-bit read/write-one-to-clear register located at `0xE000ED2A`.

Secure software can access the Non-secure version of this register via UFSR_NS located at `0xE002ED2A`. The location `0xE002ED2A` is RES0 to software executing in Non-secure state and the debugger.

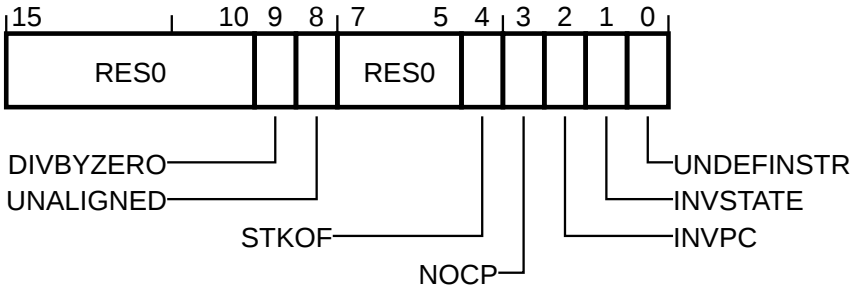
This register is banked between Security states.

This register is part of CFSR.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The UFSR bit assignments are:




**Bits [15:10]**
Reserved, RES0.

**DIVBYZERO, bit [9]**
Divide by zero flag. Sticky flag indicating whether an integer division by zero error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**UNALIGNED, bit [8]**
Unaligned access flag. Sticky flag indicating whether an unaligned access error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**Bits [7:5]**
Reserved, RES0.

**STKOF, bit [4]**
Stack overflow flag. Sticky flag indicating whether a stack overflow error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**NOCP, bit [3]**
No coprocessor flag. Sticky flag indicating whether a coprocessor disabled or not present error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVPC, bit [2]**
Invalid PC flag. Sticky flag indicating whether an integrity check error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**INVSTATE, bit [1]**
Invalid state flag. Sticky flag indicating whether an EPSR.T, EPSR.IT, or FPSCR.LTPSIZE validity error has occurred.

The possible values of this bit are:

**0**
Error has not occurred.

**1**
Error has occurred.

This bit resets to zero on a Warm reset.

**UNDEFINSTR, bit [0]**

UNDEFINED instruction flag. Sticky flag indicating whether an UNDEFINED instruction error has occurred.

The possible values of this bit are:

**0**

Error has not occurred.

**1**

Error has occurred.

This includes attempting to execute an UNDEFINED instruction associated with an enable coprocessor.

This bit resets to zero on a Warm reset.

### D1.2.264 VPR, Vector Predication Status and Control Register

The VPR characteristics are:

**Purpose**
Holds the per element predication flags.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**
Present only if MVE are implemented.

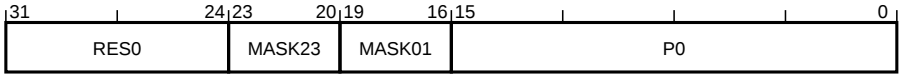
This register is RES0 if MVE are not implemented.

**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

## Field descriptions

The VPR bit assignments are:




**Bits [31:24]**
Reserved, RES0.

**MASK23, bits [23:20]**
The VPT mask bits for beat 2 and 3.

The possible values of this field are:

**0b0000**
Not in a VPT block.

**0b1000**
In a VPT block which is valid for one more instruction. The predicate flags are not inverted.

**0bx100**
In a VPT block which is valid for two more instructions. If set, the x bit causes the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block.

**0bxx10**
In a VPT block which is valid for three more instructions. If set, the x bits cause the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block.

**0bxxx1**
In a VPT block which is valid for four more instructions. If set, the x bits cause the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block.

If the PE executes a single beat per architecture tick, this field and the associated predicate flags are updated after beat 3 completes.

This field resets to an UNKNOWN value on a Warm reset.

**MASK01, bits [19:16]**
The VPT mask bits for beat 0 and 1.

The possible values of this field are:

**0b0000**
Not in a VPT block.

**0b1000**
VPT predication valid for one more instruction. The predicate flags are not inverted.

**0bx100**
In a VPT block which is valid for two more instructions. If set, the x bit causes the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block.

**0bxx10**
In a VPT block which is valid for three more instructions. If set, the x bits cause the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block.

**0bxxx1**
In a VPT block which is valid for four more instructions. If set, the x bits cause the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block.

If the PE executes a single beat per architecture tick, this field and the associated predicate flags are updated after beat 1 completes.

This field resets to an UNKNOWN value on a Warm reset.

**P0, bits [15:0]**
Predication bits. Each group of 4 bits determines the predication of each of the 4 bytes within the corresponding beat, regardless of instruction data type. See the relevant instruction descriptions and pseudocode for information on how the predication affects execution.

The possible values of this field are:

**0**
The corresponding vector lane will be masked.

**1**
The corresponding vector lane will be active.

Unprivileged access to this field is permitted, see VMRS and VMSR instructions.

This field resets to an UNKNOWN value on a Warm reset.

### D1.2.265 VTOR, Vector Table Offset Register

The VTOR characteristics are:

**Purpose**
Holds the vector table address for the selected Security state.

**Usage constraints**
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

**Configurations**
This register is always implemented.

**Attributes**
32-bit read/write register located at `0xE000ED08`.

To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

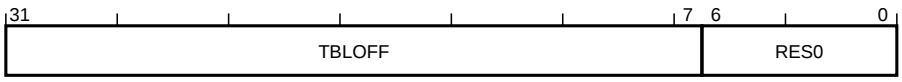
Secure software can access the Non-secure version of this register via VTOR_NS located at `0xE002ED08`. The location `0xE002ED08` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

From Armv8.1-M onwards it is IMPLEMENTATION DEFINED whether a debugger write to this register is ignored when the PE is not in Debug state.

## Field descriptions

The VTOR bit assignments are:




**TBLOFF, bits [31:7]**
Table offset. Bits [31:7] of the vector table address for the selected Security state.

It is IMPLEMENTATION DEFINED whether any of the TBLOFF bits are WI.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

**Bits [6:0]**
Reserved, RES0.

## D1.2.266 XPSR, Combined Program Status Registers

The XPSR characteristics are:

**Purpose**
Provides access to a combination of the APSR, EPSR and IPSR.

**Usage constraints**
Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.

**Configurations**
This register is always implemented.

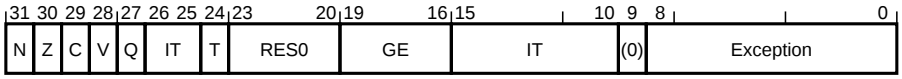
**Attributes**
32-bit read/write special-purpose register.

This register is not banked between Security states.

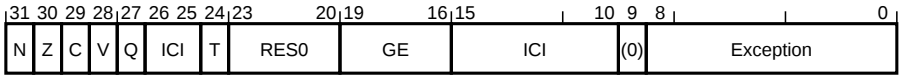
### Field descriptions

The XPSR bit assignments are:

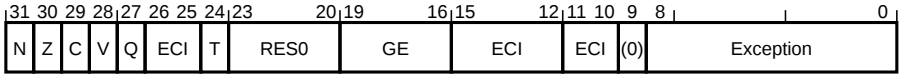
**When {XPSR[26:25], XPSR[11:10]} != 0**:




**When {XPSR[26:25], XPSR[11:10]} == 0, and a multi-cycle load or store instruction is in progress**:




**When {XPSR[26:25], XPSR[11:10]} == 0, and more than one beat-wise vector instruction is in progress**:




**N, bit [31]**
Negative flag. Reads or writes the current value of APSR.N.

**Z, bit [30]**
Zero flag. Reads or writes the current value of APSR.Z.

**C, bit [29]**
Carry flag. Reads or writes the current value of APSR.C.

**V, bit [28]**
Overflow flag. Reads or writes the current value of APSR.V.

**Q, bit [27]**
Saturate flag. Reads or writes the current value of APSR.Q.

**T, bit [24]**
T32 state. Reads or writes the current value of EPSR.T.

**Bits [23:20]**
Reserved, RES0.

**GE, bits [19:16]**
Greater-than or equal flag. Reads or writes the current value of APSR.GE.

**IT, bits [15:10,26:25]** , when [{XPSR[26:25], XPSR[11:10]} != 0]
If-then flags. Reads or writes the current value of EPSR.IT.

**ICI, bits [26:25,15:10]** , when [{XPSR[26:25], XPSR[11:10]} == 0, and a multi-cycle load or store instruction is in progress]
Interrupt continuation flags. Reads or writes the current value of EPSR.ICI.

**ECI, bits [26:25, 11:10, 15:12]** , when [{XPSR[26:25], XPSR[11:10]} == 0, and more than one beat-wise vector instruction is in progress]
Exception continuation flags for beat-wise vector instructions. Reads or writes the current value of EPSR.ECI.

**Bit [9]**
Reserved, RES0.

**Exception, bits [8:0]**
Exception number. Reads or writes the current value of IPSR.Exception.

# Part E
## Armv8-M Pseudocode

# Chapter E1
# Arm Pseudocode Definition

This chapter provides a definition of the pseudocode that this manual uses, and defines some *built-in* functions that the pseudocode uses. It contains the following sections:



**Note**

This chapter is not a formal language definition for the pseudocode. It is a guide to help understand the use of Arm pseudocode.

# E1.1 About the Arm pseudocode

The Arm pseudocode provides precise descriptions of some areas of the Arm architecture. This includes description of the decoding and operation of all valid instructions.

The following sections describe the Arm pseudocode in detail:

E1.2 *Data types* on page 1754.

E1.3 *Operators* on page 1760.

E1.4 *Statements and control structures* on page 1766.

E1.5 *Built-in functions* on page 1771 describes some built-in functions that the pseudocode functions use that this manual describes elsewhere.

E1.6 *Arm pseudocode definition index* on page 1774 contains the indexes to the pseudocode.

## E1.1.1 General limitations of Arm pseudocode

Because of the limitations inherent in all pseudocode, the Arm pseudocode and pseudocode comments describe only one particular implementation of the architecture. There are several instances where a rule relaxes the behavior that a particular piece of pseudocode describes.

The pseudocode statements `EndOfInstruction()`, `SEE`, `UNDEFINED`, `CONSTRAINED_UNPREDICTABLE`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of the statements is encountered:

- `CONSTRAINED_UNPREDICTABLE`, and `UNPREDICTABLE` mean earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed. No subsequent behavior that the pseudocode indicates occurs.
- `EndOfInstruction()`,`SEE`, and `UNDEFINED` mean that the pseudocode will terminate execution of the current instruction and pseudocode execution continues from the exception catch.

For more information, see E1.4.5 *Special statements* on page 1769.

# E1.2 Data types

This section describes:



## E1.2.1 General data type rules

Arm architecture pseudocode is a strongly typed language. Every literal and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- Tuple.
- Struct.
- Array.

The syntax of a literal determines its type. A variable can be assigned to without an explicit declaration. The variable implicitly has the type of the assigned value. For example, the following assignments implicitly declare the variables `x`, `y` and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

```
x = 1;
y = '1';
z = TRUE;
```

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. The following example declares explicitly that a variable named `count` is an integer.

```
integer count;
```

This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

## E1.2.2 Bitstrings

This section describes the bitstring data type.

### Syntax

**bits**(N)

The type name of a bitstring of length ‘N‘.

**bit**

A synonym of `bits(1)`.

### Description

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 0.

Bitstring constants literals are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants literals of type **bit** are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

The bits in a bitstring are numbered from left to right *N*-1 to 0. This numbering is used when accessing the bitstring using bitslices. In conversions to and from integers, bit *N*-1 is the MSByte and bit 0 is the LSByte. This order matches the order in which bitstrings derived from encoding diagrams are printed.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length `N` is bit (*N-1*) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings that are derived from encoding diagrams, this order matches the way that they are printed.

Bitstrings are the only concrete data type in pseudocode, corresponding directly to the contents values that are manipulated in registers, memory locations, and instructions. All other data types are abstract.

## E1.2.3 Integers

This section describes the data type for integer numbers.

### Syntax

**integer**

The type name for the integer data type.

### Description

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as integers.

Integer literals are normally written in decimal form, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal form, such as `0x55` or `0x80000000`. Hexadecimal integer literals are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If $-2^{31}$ needs to be written in hexadecimal, it must be written as `-0x80000000`.

## E1.2.4 Reals

This section describes the data type for real numbers.

### Syntax

`real`

The type name for the real data type.

### Description

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, and the pseudocode provides functions to interpret those bitstrings as reals.

Real constant literals are written in decimal form with a decimal point. This means `0` is an integer constant literal, but `0.0` is a real constant literal.

## E1.2.5 Booleans

This section describes the boolean data type.

### Syntax

**`boolean`**

The type name for the boolean data type.

`TRUE`, `FALSE`

The two values a boolean variable can take.

### Description

A boolean is a logical `TRUE` or `FALSE` value.

**Note**

This is not the same type as **`bit`**, which is a bitstring of length 1. A boolean can only take on one of two values: `TRUE` or `FALSE`.

## E1.2.6 Enumerations

This section describes the enumeration data type.

### Syntax and examples

`enumeration`

Keyword to define a new enumeration type.

```
enumeration Example {Example_One, Example_Two, Example_Three};
```

A definition of a new enumeration that is called Example, which can take on the values `Example_One`, `Example_Two`, `Example_Three`.

### Description

An enumeration is a defined set of named values.

An enumeration must contain at least one named value. A named value must not be shared between enumerations.

Enumerations must be defined explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the named values to it. By convention, each named value starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the named values are its possible *values*.

## E1.2.7 Structures

This section describes the structure data type.

### Syntax and examples

`type`

The keyword that is used to declare the structure data type.

```
type ShiftSpec is (bits(2)shift, integer amount):
```

An example definition for a new structure that is called ‘ShiftSpec‘ that contains a bitstring member that is called ‘shift‘ and an integer member called ‘amount‘. Structure definitions must not be terminated with a semicolon.

```
ShiftSpec abc;
```

A declaration of a variable that is named ‘abc‘ of type ‘ShiftSpec‘.

```
abc.shift
```

Syntax to refer to the individual members within the structure variable.

### Description

A structure is a compound data type composed of one or more data items. The data items can be of different data types. This can include compound data types. The data items of a structure are called its members and are named.

In the syntax section, the example defines a structure that is called `ShiftSpec` with two members. The first is a bitstring of length 2 named `shift` and the second is an integer that is named `amount`. After declaring a variable of that type that is named `abc`, the members of this structure are referred to as `abc.shift` and `abc.amount`.

Every definition of a structure creates a different type, even if the number and type of their members are identical. For example:

```
type ShiftSpec1 is (bits(2)shift, integer amount)

type ShiftSpec2 is (bits(2)shift, integer amount)
```

`ShiftSpec1` and `ShiftSpec2` are two different types despite having identical definitions. This means that the value in a variable of type `ShiftSpec1` cannot be assigned to variable of type `ShiftSpec2`.

#### E1.2.7.1 _Type and _Type

This subsection describes the data structure types for a particular register or payload.

### Example

```
RETPSR_Type
```

The data structure of type RETPSR.

### Description

By convention `_Type` declares a structure data type for a specific register or payload.

See the individual register descriptions for the fields that apply to a particular data structure.

## E1.2.8 Tuples

This section describes the tuple data type.

### Examples

```
(bits(32)shifter_result, bit shifter_carry_out)
```

An example of the tuple syntax.

```
(shift_t, shift_n)= ('00',0);
```

An example of assigning values to a tuple.

### Description

A tuple is an ordered set of data items, which are separated by commas and enclosed in parentheses. The items can be of different types and a tuple must contain at least one data item.

Tuples are often used as the return type for functions that return multiple results. For example, in the syntax section, the example tuple is the return type of the function `Shift_C()`, which performs a standard A32/T32 shift or rotation. Its return type is a tuple containing two data items, with the first of type, and **bits**`(32)` the second of type **bit**.

Each tuple is a separate compound data type. The compound data type is represented as a comma-separated list of ordered data types between parentheses. This means that the example tuple at the start of this section is of type `(`**bits**`(32), `**bit**`)`. The general principle that types can be implied by an assignment extends to implying the type of the elements in the tuple. For example, in the syntax section, the example assignment implicitly declares:

- `shift_t` to be of type **bits**`(2)`.
- `shift_n` to be of type **integer**.
- `(shift_t, shift_n)` to be a tuple of type `(`**bits**`(2), `**integer**`)`.

## E1.2.9 Arrays

This section describes the array data type.

### Syntax

```
array
```

The type name for the array data type.

```
array data_type array_name[A..B];
```

```
array [A...B] of data_type array_name
```

Declaration of an array of type ‘data_type‘, which might be compound data type. It is named ‘array_name‘ and is indexed with an integer range from ‘A‘ to ‘B‘.

## Description

An array is an ordered set of fixed size containing items of a single data type. This can include compound data types. Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then . . , then the upper inclusive end of the range.

For example:

The following example declares an array of 31 bitstrings of length 64, indexed from 0-30.

```
1         array bits(64) _R[0..30];
```

Arrays are always explicitly declared, and there is no notation for a constant literal array. Arrays always contain at least one element data item, because:

- Enumerations always contain at least one symbolic constant named value.
- Integer ranges always contain at least one integer.

An array declared with an enumeration type as the index must be accessed using enumeration values of that enumeration type. An array declared with an integer range type as the index must be accessed using integer values from that inclusive range. Accessing such an array with an integer value outside of the range is a coding error.

Pseudocode can also contain array-like functions such as `R[i]`, `MemU[address, size]`, or `Elem[vector, i, size]`. These functions package up and abstract additional operations that are normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing. See E1.4.2 *Function and procedure calls* on page 1766.

# E1.3 Operators

This section describes:



## E1.3.1 Relational operators

The following operations yield results of type **boolean**.

### Equality and non-equality

If two variables `x` and `y` are of the same type, their values can be tested for equality by using the expression `x == y` and for non-equality by using the expression `x != y`. In both cases, the result is of type **boolean**.

Both `x` and `y` must be of type **bits**`(N)`, `real`, `enumeration`, **boolean**, or **integer**. Named values from an `enumeration` can only be compared if they are both from the same `enumeration`. An exception is that a bitstring can be tested for equality with an integer to allow a `d==15` test.

A special form of comparison is defined with a bitstring literal that can contain bit values `'0'`, `'1'`, and `'x'`. Any bit with value `'x'` is ignored in determining the result of the comparison. For example, if `opcode` is a 4-bit bitstring, the expression `opcode == '1x0x'` matches the values `1000`, `1100`, `1001`, and `1101`. This is known as a bitmask.

> **Note**
>
> This special form is permitted in the implied equality comparisons in the **when** parts of **case** `... of ...` structures.

### Comparisons

If `x` and `y` are integers or reals, then `x < y`, `x <= y`, `x > y`, and `x = y` are less than, less than or equal, greater than, and greater than or equal comparisons between them, producing boolean results.

#### E1.3.1.1 Set membership with IN

`<expression>` **IN** `{<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions that are separated by commas.

## E1.3.2 Boolean operators

If `x` is a boolean expression, then `!x` is its logical inverse.

If `x` and `y` are boolean expressions, then `x` **&&** `y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

**Note**

This is known as short circuit evaluation.

If `x` and `y` are **boolean**s, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

**Note**

If `x` and `y` are **boolean**s or boolean expressions, then the result of `x != y` is the same as the result of exclusive-ORing `x` and `y` together. The operator `EOR` only accepts bitstring arguments.

### E1.3.3 Bitstring operators

The following operations can be applied only to bitstrings.

#### Logical operations on bitstrings

If `x` is a bitstring, `NOT(x)` is the bitstring of the same length that is obtained by logically inverting every bit of `x`.

If `x` and `y` are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length that is obtained by logically ANDing, logically ORing, and exclusive-ORing corresponding bits of `x` and `y` together.

#### Bitstring concatenation and slicing

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

The bitstring slicing operator addresses specific bits in a bitstring. This can be used to create a new bitstring from extracted bits or to set the value of specific bits. Its syntax is`x <integer_list>`, where `x` is the integer or bitstring being sliced, and `<integer_list>` is a comma-separated list of integers that are enclosed in angle brackets. 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:

- $\geq 0$.
- `< Len(x)` if `x` is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `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, and `y` is the unique integer in the range `0` to`2^(i+1)-1` that is congruent to `x` modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

The notation for a range expression is `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` represents `instr<31, 30, 29, 28>`.

`x<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>` as named bits can be referred to with the same syntax as referring to members of a struct. A

comma-separated list of named bits enclosed in angle brackets following the register name allows multiple bits to be addressed simultaneously.

For example, `APSR.<N, C, Q>` is synonymous with `APSR <31, 29, 27>`.

### E1.3.4 Arithmetic operators

Most pseudocode arithmetic is performed on integer or real values, with operands obtained by conversions from bitstrings and results that are converted back to bitstrings. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

#### Unary plus and minus

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed. Both are of the same type as `x`.

#### Addition and subtraction

If `x` and `y` are integers or reals, `x + y` and `x - y` are their sum and difference. Both are of type **`integer`** if `x` and `y` are both of type **`integer`**, and `real` otherwise.

There are two cases where the types of `x` and `y` can be different. A bitstring and an integer can be added together to allow the operation `PC + 4`. An integer can be subtracted from a bitstring to allow the operation `PC - 2`.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x)= Len(y)`, then `x + y` and `x - y` are the least significant *N* bits of the results of converting `x` and `y` to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
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`.

#### Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type **`integer`** if `x` and `y` are both of type **`integer`**, and `real` otherwise.

#### Division and modulo

If `x` and `y` are reals, then `x/y` is the result of dividing `x` by `y`, and is always of type `real`.

If `x` and `y` are integers, then `x 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 of `x/y`, `x MOD y`, or `x DIV y` in any context where `y` can be zero.

#### Scaling

If `x` and `n` are of type **`integer`**, then:

- `x << n = RoundDown(x * 2^n)`.
- `x >> n = RoundDown(x * 2^(-n))`.

### Raising to a power

If `x` is an integer or a real and `n` is an integer, then `x^n` is the result of raising `x` to the power of `n`, and:

- If `x` is of type **`integer`** then `x^n` is of type **`integer`**.
- If `x` is of type `real` then `x^n` is of type `real`.

## E1.3.5 The assignment operator

The assignment operator is the = character, which assigns the value of the right-hand side to the left-hand side. An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

This following subsection defines valid expression syntax.

### General expression syntax

An expression is one of the following:

- A literal.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register that is defined in an Arm architecture specification defines a correspondingly named pseudocode bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as '0' and ignore writes.

An expression like **`bits`**`(32)` `UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not:

- Return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE or CONSTRAINED UNPREDICTABLE and do not return UNKNOWN values,
- Be promoted as providing any useful information to software.

  **Note**

  UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment:

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates on is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

**Note**

If the right-hand side in an assignment is a function returning a tuple, an item in the assignment destination can be written as - to indicate that the corresponding item of the assigned tuple value is discarded. For example:

```
(shifted, -)= LSL_C(operand, amount);
```

The expression on the right-hand side itself can be a tuple. For example:

```
(x, y)= (function_1(), function_2());
```

Every expression has a data type.

- For a literal, this data type is determined by the syntax of the literal.
- For a variable, there are the following possible sources for the data type
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member.

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.
- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

### E1.3.6 Precedence rules

The precedence rules for expressions are:

1. Literals, variables, and function invocations are evaluated with higher priority than any operators using their results, but see E1.3.2 *Boolean operators* on page 1760.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but do not need to be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 &&j > 0 &&k > 0` is acceptable, but `i > 0 &&j > 0 ||k > 0` is not.

### E1.3.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type **`boolean`**, then **`if`** `t` **`then`** `x` **`else`** `y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

### E1.3.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator `+` has forms that act on various combinations of integers, reals and bitstrings.

Table E1-1 summarizes the operand types valid for each unary operator and the result type. Table E1-2 summarizes the operand types valid for each binary operator and the result type.

***Table E1-1, Result and operand types that are permitted for unary operators.***

| Operator | Operand Type | Result Type |
|---|---|---|
| - | integer | interger |
| | real | real |
| NOT | bits(N) | bits(N) |
| ! | boolean | boolean |

***Table E1-2, Result and operand types that are permitted for binary operators.***

| Operator | First operand type | Second operand type | Result type |
|---|---|---|---|
| == | bits(N) | integer | boolean |
| | bits(N) | bits(N) | |
| | integer | integer | |
| | real | real | |
| | enumeration | enumeration | |
| | boolean | boolean | |
| != | bits(N) | bits(N) | boolean |
| | integer | integer | |
| | real | real | |
| <, > <=, >= | integer | integer | boolean |
| | real | real | |
| +, - | integer | integer | integer |
| | real | real | real |
| | bits(N) | bits(N) | bits(N) |
| | bits(N) | integer | |
| «,» | integer | interger | integer |
| * | integer | integer | integer |
| | real | real | real |
| | bits(N) | bits(N) | bits(N) |
| / | real | real | real |
| DIV | integer | integer | integer |
| MOD | integer | integer | integer |
| | bits(N) | integer | |
| &&, \|\| | boolean | boolean | boolean |
| AND, OR, EOR | bits(N) | bits(N) | bits(N) |
| ^ | integer | integer | integer |
| | real | integer | real |

# E1.4 Statements and control structures

This section describes the statements and program structures available in the pseudocode.

## E1.4.1 Statements and Indentation

A simple statement is either an assignment, a function call, or a procedure call. Each statement must be terminated with a semicolon.

Indentation normally indicates the structure in compound statements. The statements that are contained in structures such as **`if... then... else...`** or procedure and function definitions are indented more deeply than the statement structure itself. The end of a compound statement structure and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level. Standard indentation uses four spaces for each level of indent.

## E1.4.2 Function and procedure calls

This section describes how functions and procedures are defined and called in the pseudocode.

### Procedure and function definitions

A procedure definition has the form:

```
<procedure name>(<argument prototypes>)
<statement 1>;
<statement 2>;
...
<statement n>;
```

where `<argument prototypes>` consists of zero or more argument definitions, which are separated by commas. Each argument definition consists of a type name followed by the name of the argument.

**Note**

This first definition line is not terminated by a semicolon. This distinguishes it from a procedure call.

A function definition is similar, but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
<statement 1>;
<statement 2>;
...
<statement n>;
```

Array-like functions are similar, but are written with square brackets and have two forms. These two forms exist because reading from and writing to an array element require different functions. They are frequently used in memory operations. An array-like function definition with a return type is equivalent to reading from an array. For example:

```
<return type> <function name>[<argument prototypes>]
<statement 1>;
<statement 2>;
...
<statement n>;
```

Its related function definition with no return type is equivalent to writing to an array. For example:

```
<function name>[<argument prototypes>] =<value prototype>
<statement 1>;
<statement 2>;
...
<statement n>;
```

The value prototype determines what data type can be written to the array. The two related functions must share the same name, but the value prototype and return type can be different.

### Procedure calls

A procedure call has the form:

```
    <procedure_name>(<arguments>);
```

### Return statements

A procedure return has the form:

```
return;
```

A function return has the form:

```
    return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

## E1.4.3 Conditional control structures

This section describes how conditional control structures are used in the pseudocode.

### `if...then...else...`

In addition to being a ternary operator, a multi-line `if...then...else...` structure can act as a control structure and has the form:

```
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 **`elsif`** blocks can be used.

The block of lines consisting of **`else`** and its indented statements is optional.

Abbreviated one-line forms can be used when the **`then`** part, and in the **`else`** part if it is present, contain only simple statements such as:

```
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 semi-colons. This and > the fact that the **`else`** part is optional distinguish its use as a > control structure from its use as a ternary operator.

#### `case...of...`

A `case...of...` structure has the form:

```
case <expression> of
when <literal values1>
<statement 1>;
<statement 2>;
...
<statement n>;

when <literal values2>
<statement 1>;
<statement 2>;
...
<statement n>;

...more "when" groups if required...

otherwise
<statement A>;
<statement B>;
...
<statement Z>;
```

In this structure, `<literal values1>` and `<literal values2>` consist of literal values of the same type as `<expression>`, separated by commas. There can be additional **`when`** groups in the structure. Abbreviated one line forms of **`when`** and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, the literal values can also include bitstring literals containing `'x'` bits, known as bitmasks. For details, see Equality and non-equality.

### E1.4.4 Loop control structures

This section describes the three loop control structures that are used in the pseudocode.

#### `repeat...until...`

A `repeat...until...` structure has the form:

```
repeat
<statement 1>;
<statement 2>;
...
<statement n>;
until <boolean_expression>;
```

It executes the statement block at least once, and the loop repeats until `<`**`boolean`** `expression>` evaluates to `TRUE`. Variables explicitly declared inside the loop body have scope local to that loop and might not be accessed outside the loop body.

#### `while...do`

A `while...do` structure has the form:

```
while <boolean_expression> do
<statement 1>;
<statement 2>;
...
statement n>;
```

It begins executing the statement block only if the boolean expression is true. The loop then runs until the expression is false.

#### for...

A **for**... structure has the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
<statement 1>;
<statement 2>;
...
<statement n>;
```

The <assignable_expression> is initialized to <integer_expr1> and compared to <integer_expr2>. If <integer_expr1> is less than <integer_expr2>, the loop body is executed and the <assignable_expression >incremented by one. This repeats until <assignable expression> is more than or equal to <integer_expr2>.

There is an alternate form:

```
for <assignable_expression> = <integer_expr1> downto <integer_expr2>
```

where <integer_expr1> is decremented after the loop body executes and continues until <assignable expression> is less than or equal than <integer_expr2>.

#### Try...Catch

A try...catch structure has the following form:

```
try
<statement 1>;
<statement 2>;
...
<statement n>;

catch <exception>
<statement a>;
<statement b>;
...
<statement z>;
```

The purpose of this structure is to catch exceptions that are generated by the try statements.

### E1.4.5 Special statements

This section describes statements with particular architecturally defined behaviors.

#### UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a pseudocode exception that will be caught by the try...catch block. When caught, this might result in an UNDEFINSTR UsageFault, NOP or NOCP UsageFault.

#### UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior that is defined by the current pseudocode, apart from behavior that is required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

#### CONSTRAINED UNPREDICTABLE

This subsection describes the statement:

```
CONSTRAINED_UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior that is defined by the current pseudocode, apart from behavior that is required to determine that the special case applies. The replacement behavior is CONSTRAINED UNPREDICTABLE within the limits defined for each particular case, and might vary.

#### SEE...

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior that is defined by the current instruction pseudocode, apart from behavior that is required to determine that the special case applies.

It usually refers to another instruction, but can also refer to another encoding or note of the same instruction.

#### IMPLEMENTATION DEFINED

This subsection describes the statement:

```
IMPDEF {"<text>"};
```

This statement indicates a special case that provides an IMPLEMENTATION DEFINED value or behavior. An optional `<text>` field can give more information.

### E1.4.6 Comments

The pseudocode supports two styles of comments:

- `//` starts a comment that is terminated by the end of the line.
- `/*` starts a comment that is terminated by `*/` .

`/**/` statements might not be nested, and the first `*/` ends the comment.

**Note**

Comment lines do not require a terminating semicolon.

# E1.5 Built-in functions

This section describes:



## E1.5.1 Bitstring manipulation functions

The following bitstring manipulation functions are defined:

### Bitstring length

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.

### Bitstring concatenation and replication

If `x` is a bitstring and `n` is an integer with `n >= 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of `n` copies of `x` concatenated together.
- `Zeros(n) = Replicate('0', n)`.
- `Ones(n) = Replicate('1', n)`.

### Bitstring count

If `x` is a bitstring, `BitCount(x)` is an integer result equal to the number of bits of `x` that are ones.

### Testing a bitstring for being all zero or all ones

If `x` is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of `x` are zeros and `FALSE` if any of them are ones
- `IsZeroBit(x)` produces `'1'` if all of the bits of `x` are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnit(x)` work in the corresponding ways. This means:

```
1   IsZero(x) = (BitCount(x) == 0)
2   IsOnes(x) = (BitCount(x) == Len(x))
3   IsZeroBit(x) = if IsZero(x) then '1' else '0'
4   IsOnit(x) = if IsOnes(x) then '1' else '0'
```

### Lowest and highest set bits of a bitstring

If `x` is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of the bits of `x` that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to *N*. This means:

  `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.

- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to *N*-1. This means:

  `CountLeadingSignBits(x)= CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>).`

### Zero-extension and sign-extension of bitstrings

If `x` is a bitstring and `i` is an integer, then `ZeroExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i)= x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i)=Replicate('0', i-Len(x)): x
```

If `x` is a bitstring and `i` is an integer, then `SignExtend(x, i)` is `x` extended to a length of `i` bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i)= x`, and if `i > Len(x)`, then:

```
SignExtend(x, i)= Replicate(TopBit(x), i-Len(x)): x
```

It is a pseudocode error to use either `ZeroExtend(x,i)` or `SignExtend(x, i)` in a context where it is possible that`i < Len(x)`.

### Converting bitstrings to integers

If `x` is a bitstring, `SInt()` is the integer whose twos complement representation is `x`.

`UInt()` is the integer whose unsigned representation is `x`.

`Int(x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument.

## E1.5.2 Arithmetic functions

This section defines built-in arithmetic functions.

### Absolute value

If `x` is either of type real or integer, `Abs(x)` returns the absolute value of `x`. The result is the same type as `x`.

### Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` so that `n <= x`.
- `RoundUp(x)` produces the smallest integer `n` so that `n >= x`
- `RoundTowardsZero(x)` produces:
  - `RoundDown(x)` if `x > 0.0`.
  - `0` if `x == 0.0`.
  - `RoundUp(x)` if `x < 0.0`.

If `x` and `y` are both of type **`integer`**, `Align(x, y)= y * (x DIV y)`, and is of type **`integer`**.

If `x` is of type `bitstring` and `y` is of type **`integer`**, `Align(x, y)= (Align(UInt(x),y))<Len(x)-1:0>`, and is a `bitstring` of the same length as `x`.

It is a pseudocode error to use either form of `Align(x, y)` in any context where `y` can be 0. In practice, `Align(x, y)` is only used with `y` a constant power of two, and the bitstring form used with `$y =2^n$` has the effect of producing its argument with its n low-order bits forced to zero.

### Maximum and minimum

If `x` and `y` are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. x and y must both be of type integer or of type real. The function returns a value of the same type as its operands.

# E1.6 Arm pseudocode definition index

This section contains the following tables:

Table E1-3 which contains the pseudocode data types.

Table E1-4 which contains the pseudocode operators.

Table E1-5 which contains the pseudocode keywords and control structures.

Table E1-6 which contains the statements with special behaviors.

### Table E1-3 Index of pseudocode data types

| Keyword | Meaning |
|---|---|
| `array` | Type name for the array type |
| `bit` | Keyword equivalent to `bits(1)` |
| `bits(N)` | Type name for the bitstring of length `N` data type |
| `boolean` | Type name for the boolean data type |
| `enumeration` | Keyword to define a new enumeration type |
| `integer` | Type name for the integer data type |
| `real` | Type name for the real data type |
| `type` | Keyword to define a new structure |

## Table E1-4 Index of pseudocode operators

| Operator | Meaning |
|---|---|
| − | Unary minus on integers or reals |
| | Subtraction of integers, reals, and bitstrings |
| | used in the left-hand side of an assignment or a tuple to discard the result |
| + | Unary plus on integers or reals |
| | Addition of integers, reals, and bitstrings |
| . | Extract named member from a list |
| | Integer in bitstring extraction operator |
| : | Bitstring concatenation |
| | Integer range in bitstring extraction operator |
| ! | Boolean NOT |
| != | Comparison for inequality |
| (...) | Around arguments of procedure or function |
| [...] | Around array index |
| | Around arguments of array-like function |
| * | Multiplication of integers, reals and bitstrings |
| / | Divsion of integers and reals (real result) |
| && | Boolean AND |
| < | *Less than* comparison of integers and reals |
| <...> | Slicing of specified bits or bitstring or integer |
| << | Mulitply integer by power of 2 (with rounding towards infinity) |
| <= | *Less than or equal* comparison of integers and reals |
| = | Assignment operator |
| == | Comparison for equality |
| > | *Greater than* comparison of integers and reals |
| >= | *Greater than or equal* comparison of integers and reals |
| >> | Divide integer by power of 2 |
| \|\| | Boolean OR |
| ^ | Exponential operator |
| AND | Bitwise AND of bitstrings |
| DIV | Quotient from integer division |
| EOR | Bitwise EOR of bitstrings |
| IN | Test membership of a certain expression in a set of values |
| MOD | Remainder from integer division |
| NOT | Bitwise inversion of bitstrings |
| OR | Bitwise OR of bitstrings |

### Table E1-5 Index of pseudocode keywords and control structures

| Operator | Meaning |
|---|---|
| `/*...*/` | Comment delimiters |
| `//` | Introduces comment terminated by end of line |
| `case...of...` | Control structure |
| `FALSE` | One of two values a boolean can take (other than `TRUE`) |
| `for...=...to...` | Loop control structure, counting up from the initial value to the upper limit |
| `for...=...downto...` | Loop control structure, counting down from the initial value to the lower limit |
| `if...then...else...` | Condition expression selecting between two values |
| `if...then...else...` | Conditional control structure |
| `otherwise` | Introduces default in `case...of...` control structure |
| `repeat...until...` | Loop control structure that runs at least once until the termination condition is satisfied |
| `return` | Procedure or function return |
| `TRUE` | One of two values a boolean can take (other than `FALSE`) |
| `try...catch` | Control structure |
| `when` | Introduces a specific case in `case...of...` control structure |
| `while...do...` | Loop control structure that runs until the termination condition is satisfied |

### Table E1-6 Index of special statements

| Keyword | Meaning |
|---|---|
| `IMPLEMENTATION_DEFINED` | Describes IMPLEMENTATION DEFINED behavior. |
| `SEE` | Points to other pseudocode to use instead |
| `UNDEFINED` | Cause Undefined Instruction exception |
| `UNKNOWN` | Unspecified value |
| `CONSTRAINED_UNPREDICTABLE` | Unspecified behavior within limits |
| `UNPREDICTABLE` | Unspecified behavior |

## E1.7 Additional functions

The following functions are not listed in E2 Pseudocode specification, and are only described in this section.

### E1.7.1 `IsSee()`

`IsSee()` returns TRUE if the exception variable that is passed to it was created because all the encodings that matched the instruction that was being decoded called SEE.

See *SEE...*.

### E1.7.2 `IsUndefined()`

`IsUndefined()` returns TRUE if the exception variable that is passed to it was created because either the instruction that was being decoded did not match any known encoding, or because one of the encodings that was matched called the special statement UNDEFINED.

See UNDEFINED.

# Chapter E2
# **Pseudocode Specification**

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

# E2.1 Alphabetical Pseudocode List

## E2.1.1 _AdvanceVPTState

```
// Advances VPT state

boolean _AdvanceVPTState;
```

## E2.1.2 _ITStateChanged

```
// Indicates a write to ITSTATE

boolean  _ITStateChanged;
```

## E2.1.3 _Mem

```
// _Mem[] - non-assignment (read) form
// ===================================
// Perform single-copy atomic, aligned, little-endian read from physical memory

(boolean, bits(8*size)) _Mem(AddressDescriptor memaddrdesc, integer size)
    assert size == 1 || size == 2 || size == 4;

// _Mem[] - assignment (write) form
// ================================
// Perform single-copy atomic, aligned, little-endian write to physical memory

boolean _Mem(AddressDescriptor memaddrdesc, integer size, bits(8*size) value)
    assert size == 1 || size == 2 || size == 4;
```

## E2.1.4 _NextInstrAddr

```
// Address of  next instruction to be fetched in case of branch type operation

bits(32) _NextInstrAddr;
```

## E2.1.5 _NextInstrITState

```
// Updated ITSTATE for next instruction

ITSTATEType _NextInstrITState;
```

## E2.1.6 _PCChanged

```
// Indicates a change in instruction fetch address due to branch type operations

boolean  _PCChanged;
```

## E2.1.7 _PendingReturnOperation

```
// Indicate any pending exception returns

boolean _PendingReturnOperation;
```

## E2.1.8 _RName

```
// 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) _RName[RNames];
```

### E2.1.9 _S

```
// The 32-bit extension register bank for the FP extension.

array bits(32) _S[0..31];
```

### E2.1.10 _SP

```
// _SP()
// =====

// Non-assignment form

bits(32) _SP(RNames spreg)
    assert ( (spreg == RNamesSP_Main_NonSecure)                      ||
             ((spreg == RNamesSP_Main_Secure)    && HaveSecurityExt()) ||
              (spreg == RNamesSP_Process_NonSecure)                  ||
             ((spreg == RNamesSP_Process_Secure) && HaveSecurityExt()) );

    return _RName[spreg]<31:2>:'00';

// Assignment form

ExcInfo _SP(RNames spreg, boolean excEntry, boolean skipLimitCheck, bits(32) value)
    assert ( (spreg == RNamesSP_Main_NonSecure)                      ||
             ((spreg == RNamesSP_Main_Secure)    && HaveSecurityExt()) ||
              (spreg == RNamesSP_Process_NonSecure)                  ||
             ((spreg == RNamesSP_Process_Secure) && HaveSecurityExt()) );

    excInfo = DefaultExcInfo();
    if !skipLimitCheck && ViolatesSPLim(spreg, value) then
        isSecure = ((spreg == RNamesSP_Main_Secure) ||
                    (spreg == RNamesSP_Process_Secure));
        // If the stack limit is violated during exception entry then the stack
        // pointer is set to the limit value. This both prevents violations and
        // ensures that the stack pointer is 8 byte aligned.
        if excEntry then
            _RName[spreg] = LookUpSPLim(spreg);

        // Raise the appropriate exception and syndrome information
        if isSecure then
            UFSR_S.STKOF  = '1';
        else
            UFSR_NS.STKOF = '1';
        // Create the exception. NOTE: If Main Extension is not implemented the fault always
        // escalates to HardFault.
        excInfo = CreateException(UsageFault, TRUE, isSecure);
        if !excEntry then
            HandleException(excInfo);
    else
        // Stack pointer only updated normally if limit not violated
        _RName[spreg] = value<31:2>:'00';
    return excInfo;
```

### E2.1.11 abs

```
// Abs()
// =====

```

```
__overloaded integer Abs(integer x)
    return if x >= 0 then x else -x;

__overloaded real Abs(real x)
    return if x >= 0.0 then x else -x;
```

### E2.1.12 AccessAttributes

```
// Memory access attributes

type AccessAttributes is (
    boolean iswrite,       // TRUE for memory stores, FALSE for load accesses
    boolean ispriv,        // TRUE if the access is privileged, FALSE if unprivileged
    AccType acctype
)
```

### E2.1.13 AccType

```
// Memory reference access type

enumeration AccType { AccType_NORMAL,       // Normal loads and stores
                      AccType_MVE,          // Loads and stores generated by MVE instructions
                      AccType_ORDERED,      // Load-Acquire and Store-Release
                      AccType_STACK,        // HW generated stacking / unstacking operation
                      AccType_LAZYFP,       // HW generated stacking due to lazy
                                            // floating point state preservation
                      AccType_IFETCH,       // Instruction fetch
                      AccType_DBG,          // Loads and Stores generated by the Debugger
                      AccType_VECTABLE      // Vector table fetch
                    };
```

### E2.1.14 ActivateException

```
// ActivateException()
// ===================

ActivateException(integer exceptionNumber, boolean excIsSecure)
    // If the exception is Secure, directly entry the Secure state.
    CurrentState = if excIsSecure
                   then SecurityState_Secure else SecurityState_NonSecure;
    IPSR.Exception = exceptionNumber<8:0>;         // Update IPSR to this exception. This
        also
                                                   // causes a transition to privileged
                                                       handler
                                                   // mode as IPSR.Exception != 0
    if HaveMainExt() then
        ITSTATE = Zeros(8);                        // IT/ICI/ECI bits cleared
    for i = 0 to MAX_OVERLAPPING_INSTRS-1
        _InstInfo[i].Valid = FALSE;
    // PRIMASK, FAULTMASK, BASEPRI unchanged on exception entry
    if HaveMveOrFPExt() then
        CONTROL.FPCA   = '0';                      // Floating-point Extension only
        CONTROL_S.SFPA = '0';
    CONTROL.SPSEL = '0';                           // CONTORL.SPSEL is updated to indicate
        the
                                                   // selection of the Main stack pointer -
                                                       SP_main
                                                   // CONTROL.nPRIV unchanged
    // Transition exception from pending to active
    SetPending(exceptionNumber, excIsSecure, FALSE);
    SetActive(exceptionNumber, excIsSecure, TRUE);
```

### E2.1.15 ActiveFPState

```
// ActiveFPState()
// ===============

boolean ActiveFPState()
    // Is the FP state accessible
    (active, -) = IsCPEnabled(10);

    // Is FP lazy state preservation active
    if active then
        if FPCCR_S.S == '1' then
            lspact = FPCCR_S.LSPACT;
        else
            lspact = FPCCR_NS.LSPACT;
        active = lspact == '0';

    // Check ASPEN to determine if the PE or software is managing the FP state
    if active && FPCCR.ASPEN == '1' then
        // If the PE is managing the FP state then FPCA can be used to indicate
        // if the current context has an active FP state. Similarly SFPA is also
        // checked to determine if the FP state is active for the Secure state.
        active = CONTROL.FPCA == '1' && (!IsSecure() || CONTROL_S.SFPA == '1');
    return active;
```

### E2.1.16 AddressDescriptor

```
// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
    bits(32)         paddress,   // Physical Address
    AccessAttributes accattrs
)
```

### E2.1.17 AddrType

```
// Indicates address type

enumeration AddrType { AddrType_NORMAL,
                       AddrType_EXC_RETURN,
                       AddrType_FNC_RETURN
                     };
```

### E2.1.18 AddWithCarry

```
// 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>;  // same value as 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);
```

### E2.1.19 AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)

    case cmode<3:1> of
        when '000'
```

```
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8);  imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8);  imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8);  imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8);  imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                UNDEFINED;

    return imm64;
```

### E2.1.20 align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

### E2.1.21 ArchVersion

```
// Indicates architecture version

enumeration ArchVersion {
    Armv8p0,
    Armv8p1
};
```

### E2.1.22 ASR

```
// 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;
```

### E2.1.23 ASR_C

```
// 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);
```

### E2.1.24 BeatComplete

```
// BeatComplete
// ============

// The BeatComplete value indicates whether the 4 beats from 2 instructions have
// been performed. The flags are packed into an 8 bit value as follows:
// bit 0: beat 0 of instruction 0
// bit 1: beat 1 of instruction 0
// ...
// bit 4: beat 0 of instruction 1
// bit 5: beat 1 of instruction 1
// ...
//
// NOTE: The beat execution rules mean that only a few flag combinations are
//       valid.

// Non-assignment form
bits(8) BeatComplete
    bits(8) beatComplete;
    case EPSR.ECI of
        when '00000000' beatComplete = '0000 0000';
        when '00000001' beatComplete = '0000 0001';
        when '00000010' beatComplete = '0000 0011';
        when '00000100' beatComplete = '0000 0111';
        when '00000101' beatComplete = '0001 0111';
        otherwise assert(FALSE);
    return beatComplete;

// Assignment form
BeatComplete = bits(8) value
    case value of
        when '0000 0000' EPSR.ECI = 0<7:0>;
        when '0000 0001' EPSR.ECI = 1<7:0>;
        when '0000 0011' EPSR.ECI = 2<7:0>;
        when '0000 0111' EPSR.ECI = 4<7:0>;
        when '0001 0111' EPSR.ECI = 5<7:0>;
        otherwise
            assert(FALSE);
```

### E2.1.25 BeatSchedule

```
// BeatSchedule()
// ==============

type InstInfoType is (
    bits(32)  Opcode,
    integer   Length,
    boolean   Valid
)

array [0..MAX_OVERLAPPING_INSTRS-1] of InstInfoType _InstInfo;
integer _InstID;
integer _BeatID;
```

### E2.1.26 BigEndian

```
// BigEndian()
// ===========

boolean BigEndian(bits(32) startAddress, integer size)
    // If AIRCR.ENDINANESS is 0 then the PE is in little endian mode
    if AIRCR.ENDIANNESS == '0' then
        return FALSE;
    // ...otherwise the PE is in big endian mode, however; the PPB
    // space (0xE0000000 to 0xE00FFFFF) is always little endian.
    endAddress     = startAddress + size;
    startPpbAccess = UInt(startAddress<31:20>) == 0xE00;
    endPpbAccess   = UInt(endAddress<31:20>)   == 0xE00;
    // If an access crosses the PPB boundary then it is
    // CONSTRAINED_UNPREDICTABLE if the PE is in big endian mode
    if startPpbAccess != endPpbAccess then
        CONSTRAINED_UNPREDICTABLE;
    return !startPpbAccess;
```

### E2.1.27 BigEndianReverse

```
// 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;
```

### E2.1.28 bitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

### E2.1.29 BitReverseShiftRight

```
// BitReverseShiftRight()
// ======================

bits(N) BitReverseShiftRight(bits(N) x, integer R)
    reversed = Zeros(N);
    if R > N then
        R = N;
    for i = 0 to R-1
        reversed<R-i-1> = x<i>;
    return reversed;
```

### E2.1.30 BranchCall

```
// BranchCall()
// ============

BranchCall(bits(32) address, boolean allowNonSecure)
    // If in the Secure state and transitions to the Non-secure state are allowed
    // then the target state is specified by the LSB of the target address
    if HaveSecurityExt() && allowNonSecure && IsSecure() then
        EPSR.T = '1';
        if address<0> == '0' then
            CurrentState = SecurityState_NonSecure;
            if HaveMveOrFPExt() then CONTROL_S.SFPA = '0';
            if HaveLOBExt() then
                LO_BRANCH_INFO.VALID = '0';
    else
        EPSR.T = address<0>;
        // If EPSR.T == 0 then an exception is taken on the next
        // instruction:  UsageFault('Invalid State') if the Main Extension is
        // implemented; HardFault otherwise

    BranchTo(address<31:1>:'0');
```

### E2.1.31 BranchReturn

```
// BranchReturn()
// ==============

ExcInfo BranchReturn(bits(32) address, boolean allowNonSecure)
    exc = DefaultExcInfo();

    case IsReturn(address) of
        when AddrType_NORMAL
            BranchCall(address, allowNonSecure);
        when AddrType_FNC_RETURN
            // Unlike exception return, any faults raised during a FNC_RETURN
            // unstacking are raised synchronously with the instruction that triggered
            // the unstacking.
            exc = FunctionReturn();
        when AddrType_EXC_RETURN
            // If enabled, the IESB contains asynchronous RAS / BusFault errors to the
            // exception context.
            if AIRCR.IESB == '1' then
                exc = SynchronizeBusFault();
            // The actual exception return is performed when the
            // current instruction completes. This is because faults that occur
            // during the exception return are handled differently from faults
            // raised during the instruction execution.
            if exc.fault == NoFault then
                PendReturnOperation(address);

    return exc;
```

### E2.1.32 BranchTo

```
// BranchTo()
// ==========

BranchTo(bits(32) address, boolean commit)
    if HaveLOBExt() then
        // Any branch between a branch future instruction and the associated
        // branch point invalidates the branch info cache
        if LO_BRANCH_INFO.VALID == '1' && LO_BRANCH_INFO.BF == '1' then
            LO_BRANCH_INFO.VALID = '0';

    // Sets the address to fetch the next instruction from. NOTE: The current PC
```

```
    // is not changed directly as this would modify the result of
    // ThisInstrAddr(), which would cause the wrong return addresses to be used
    // for some types of exception. The actual update of the PC is done in the
    // InstructionAdvance() function after the instruction finishes executing.
    _NextInstrAddr = address<31:1>:'0';
    _PCChanged     = TRUE;
    // Clear any pending exception returns
    _PendingReturnOperation = FALSE;

    if commit then
        // This directly commits the change to the PC, so ThisInstrAddr()
        // and NextInstrAddr() both point to the target address. Used for exception
        // returns and resets so the state is consistent before the next instruction
        // (or exception) is taken.
        _RName[RNamesPC] = _NextInstrAddr;


BranchTo(bits(32) address)
    BranchTo(address, FALSE);
```

### E2.1.33 BusFaultBarrier

```
// BusFaultBarrier()
// =================

// Forces any latent BusFault (both RAS and non-RAS) to be recognised.
// This function returns TRUE if a BusFault was detected.
boolean BusFaultBarrier();
```

### E2.1.34 CallSupervisor

```
// CallSupervisor()
// ================

CallSupervisor()
    excInfo = CreateException(SVCall);
    HandleException(excInfo);
```

### E2.1.35 CanDebugAccessFP

```
// CanDebugAccessFP()
// ==================

boolean CanDebugAccessFP()
    canAccessFP = (!HaveSecurityExt() || DHCSR.S_SDE == '1' ||
                   (FPCCR_S.S == '0' && NSACR.CP10 == '1'));

    // Unprivileged-only debug for the state associated with the floating-point
    // context restricts access via CPACR checking and if a lazy context is active.
    if HaveUDE() then
        if FPCCR_S.S == '1' && DHCSR.S_SUIDE == '1' then
            canAccessFP = canAccessFP && CPACR_S.CP10 == '11' && FPCCR_S.LSPACT != '1';
        elsif FPCCR_S.S == '0' && DHCSR.S_NSUIDE == '1' then
            canAccessFP = canAccessFP && CPACR_NS.CP10 == '11' && FPCCR_NS.LSPACT != '1';
    return canAccessFP;
```

### E2.1.36 CanHaltOnEvent

```
// CanHaltOnEvent()
// ================

boolean CanHaltOnEvent(boolean is_secure)
    if !HaveSecurityExt() then assert !is_secure;
    return (HaveHaltingDebug()                              &&
```

```
            !Halted                                          &&
            DHCSR.C_DEBUGEN == '1'                           &&
            (!is_secure            || DHCSR.S_SDE == '1') &&
            (HaltingDebugAllowed() || UnprivHaltingDebugAllowed(is_secure)));
```

### E2.1.37 CanPendMonitorOnEvent

```
// CanPendMonitorOnEvent()
// =======================

boolean CanPendMonitorOnEvent(boolean isSecure, boolean checkPri, boolean checkEn)
    if !HaveSecurityExt() then assert !isSecure;

    result = HaveDebugMonitor() && !CanHaltOnEvent(isSecure) && !Halted;

    if checkEn then
        result = result && ((DEMCR.MON_EN  == '1') ||
                            (HaveUDE() && DEMCR.UMON_EN == '1' && !CurrentModeIsPrivileged(
                                isSecure)));

    if isSecure then
        result = result && DEMCR.SDME == '1';

    if checkPri then
        result = result && ExceptionPriority(DebugMonitor, isSecure, TRUE) <
            ExecutionPriority();

    return result;
```

### E2.1.38 CheckCPEnabled

```
// CheckCPEnabled()
// ================

ExcInfo CheckCPEnabled(integer cp, boolean privileged, boolean secure)
    (enabled, toSecure) = IsCPEnabled(cp, privileged, secure);
    if !enabled then
        if toSecure then
            UFSR_S.NOCP  = '1';
        else
            UFSR_NS.NOCP = '1';
        excInfo = CreateException(UsageFault, TRUE, toSecure);
    else
        excInfo = DefaultExcInfo();
    return excInfo;

ExcInfo CheckCPEnabled(integer cp)
    return CheckCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());
```

### E2.1.39 CheckDecodeFaults

```
// CheckDecodeFaults()
// ===================
// Check and raise faults in the correct order for MVE and floating-point
// instructions.

CheckDecodeFaults(ExtType extType)
    // Is the instruction in the co-processer space
    (isCP, cpNumber) = IsCPInstruction(ThisInstr());
    assert(isCP);

    // Is the co-processer enabled, this may raise a NOCP UsageFault
    excInfo = CheckCPEnabled(cpNumber);
    HandleException(excInfo);

```

```
    // Check if the type of instruction is supported.
    case extType of
        when ExtType_HpFp      if MVFR1.FP16 == '0000'     then UNDEFINED;
        when ExtType_SpFp      if MVFR0.FPSP == '0000'     then UNDEFINED;
        when ExtType_DpFp      if MVFR0.FPDP == '0000'     then UNDEFINED;
        when ExtType_Mve       if MVFR1.MVE  == '0000'     then UNDEFINED;
        when ExtType_MveFp     if MVFR1.MVE  != '0010'     then UNDEFINED;
        when ExtType_MveOrFp
            // Always raises a NOCP fault if MVE and the Floating-point
            // Extension are not present
            assert(MVFR1.MVE != '0000' || MVFR0.FPSP != '0000');
        when ExtType_MveOrDpFp
            if MVFR1.MVE == '0000' && MVFR0.FPDP == '0000' then UNDEFINED;
        when ExtType_Unknown                                UNDEFINED;
        otherwise assert(FALSE);
```

### E2.1.40 CheckFPDecodeFaults

```
// CheckFPDecodeFaults()
// =====================

CheckFPDecodeFaults(bits(2) size)
    // Checks the size field and identifies the correct ExtType
    case size of
        when '00' CheckDecodeFaults(ExtType_Unknown);
        when '01' CheckDecodeFaults(ExtType_HpFp);
        when '10' CheckDecodeFaults(ExtType_SpFp);
        when '11' CheckDecodeFaults(ExtType_DpFp);
```

### E2.1.41 CheckPermission

```
// CheckPermission()
// =================

ExcInfo CheckPermission(Permissions perms, bits(32) address, AccType acctype,
                        boolean iswrite, boolean ispriv, boolean isSecure)
    if !perms.apValid then
        fault = TRUE;
    elsif (perms.xn == '1') && (acctype == AccType_IFETCH) then
        fault = TRUE;
    else
        case perms.ap of
            when '00'  fault = !ispriv;
            when '01'  fault = FALSE;
            when '10'  fault = !ispriv || iswrite;
            when '11'  fault = iswrite;
            otherwise   UNPREDICTABLE;

    // If a fault occurred generate the syndrome info and create the exception.
    if fault then
        // Create and write out the syndrome info on implementations with Main Extension.
        if HaveMainExt() then
            MMFSR_Type fsr = Zeros(8);
            case acctype of
                when AccType_IFETCH
                    fsr.IACCVIOL = '1';
                when AccType_STACK
                    if iswrite then
                        fsr.MSTKERR = '1';
                    else
                        fsr.MUNSTKERR = '1';
                when AccType_LAZYFP
                    fsr.MLSPERR   = '1';
                when AccType_NORMAL, AccType_MVE, AccType_ORDERED
                    fsr.MMARVALID = '1';
                    fsr.DACCVIOL  = '1';
                when AccType_DBG
```

```
                    // DAP errors don't set syndrome
                otherwise
                    assert(FALSE);

            // Write the syndrome information to the correct instance of banked
            // registers
            if isSecure then
                MMFSR_S = MMFSR_S OR fsr;
                if fsr.MMARVALID == '1' then
                    MMFAR_S = address;
            else
                MMFSR_NS = MMFSR_NS OR fsr;
                if fsr.MMARVALID == '1' then
                    MMFAR_NS = address;

        // Create the exception. NOTE: If Main Extension is not implemented the fault
        // escalates to a HardFault
        excInfo = CreateException(MemManage, TRUE, isSecure);
    else
        excInfo = DefaultExcInfo();
    return excInfo;
```

### E2.1.42 ClearEventRegister

```
// ClearEventRegister
// ==================
// Clears the Event register

ClearEventRegister();
```

### E2.1.43 ClearExclusiveByAddress

```
// ClearExclusiveByAddress
// =======================
// Clear the global exclusive monitor for all PEs, except for the PE specified
// by processorid for which an address region including any of size bytes
// starting from address has had a request for an exclusive access

ClearExclusiveByAddress(bits(32) address, integer exclprocessorid, integer size);
```

### E2.1.44 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear local exclusive monitor records for the PE.

ClearExclusiveLocal(integer processorid);
```

### E2.1.45 ComparePriorities

```
// ComparePriorities()
// ===================

boolean ComparePriorities(integer exc0Pri, integer exc0Number, boolean exc0IsSecure,
                          integer exc1Pri, integer exc1Number, boolean exc1IsSecure)
    if exc0Pri != exc1Pri then
        takeE0 = exc0Pri < exc1Pri;
    elsif exc0Number != exc1Number then
        takeE0 = exc0Number < exc1Number;
    elsif exc0IsSecure != exc1IsSecure then
        takeE0 = exc0IsSecure;
    else
        // The two exceptions have exactly the same priority, so exception 0
        // cannot be taken in preference to exception 1.
```

```
        takeE0 = FALSE;
    return takeE0;


boolean ComparePriorities(ExcInfo exc0Info, boolean groupPri,
                          integer exc1Pri, integer exc1Number, boolean exc1IsSecure)
    exc0Pri = ExceptionPriority(exc0Info.fault, exc0Info.isSecure, groupPri);
    return ComparePriorities(exc0Pri, exc0Info.fault, exc0Info.isSecure,
                             exc1Pri, exc1Number,     exc1IsSecure);
```

### E2.1.46 Cond

```
// Condition code definitions
// ==========================

constant bits(4) CondEQ = 0x0<3:0>;
constant bits(4) CondNE = 0x1<3:0>;
constant bits(4) CondCS = 0x2<3:0>;
constant bits(4) CondCC = 0x3<3:0>;
constant bits(4) CondMI = 0x4<3:0>;
constant bits(4) CondPL = 0x5<3:0>;
constant bits(4) CondVS = 0x6<3:0>;
constant bits(4) CondVC = 0x7<3:0>;
constant bits(4) CondHI = 0x8<3:0>;
constant bits(4) CondLS = 0x9<3:0>;
constant bits(4) CondGE = 0xA<3:0>;
constant bits(4) CondLT = 0xB<3:0>;
constant bits(4) CondGT = 0xC<3:0>;
constant bits(4) CondLE = 0xD<3:0>;
constant bits(4) CondAL = 0xE<3:0>;
constant bits(4) CondNV = 0xF<3:0>;
```

### E2.1.47 ConditionHolds

```
// ConditionHolds()
// ================

boolean ConditionHolds(bits(3) shortCond, bit n, bit z, bit c, bit v)
    // Expand the short condition to the standard 4 bit representation
    case shortCond of
        when '000' cond = CondEQ;
        when '001' cond = CondNE;
        when '010' cond = CondCS;
        when '011' cond = CondHI;
        when '100' cond = CondGE;
        when '101' cond = CondLT;
        when '110' cond = CondGT;
        when '111' cond = CondLE;
    return ConditionHolds(cond, n, z, c, v, TRUE);

boolean ConditionHolds(bits(4) cond)
    return ConditionHolds(cond, APSR.N, APSR.Z, APSR.C, APSR.V, FALSE);

boolean ConditionHolds(bits(4) cond, bit n, bit z, bit c, bit v, boolean allowNV)
    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (z == '1');               // EQ or NE
        when '001'  result = (c == '1');               // CS or CC
        when '010'  result = (n == '1');               // MI or PL
        when '011'  result = (v == '1');               // VS or VC
        when '100'  result = (c == '1') && (z == '0'); // HI or LS
        when '101'  result = (n == v);                 // GE or LT
        when '110'  result = (n == v)   && (z == '0'); // GT or LE
        when '111'  result = TRUE;                     // AL or possibly NV

    // The LSB of the condition code is an invert flag. Some situations prohibit
    // execute never, and treat it the same as execute always. This applies the
```

```
    // invert taking into account whether the inverse of always is allowed.
    if cond<0> == '1' && (cond != CondNV || allowNV)then
        result = !result;
    return result;
```

### E2.1.48 ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionPassed(CurrentCond());

boolean ConditionPassed(bits(4) cond)
    passed = ConditionHolds(cond);
    return passed;
```

### E2.1.49 ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

### E2.1.50 ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which);
```

### E2.1.51 ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a wrapper for UNPREDICTABLE cases where the constrained result is
// either TRUE or FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

### E2.1.52 ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================
// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which);
```

### E2.1.53 ConsumeExcStackFrame

```
// ConsumeExcStackFrame()
// ======================

ConsumeExcStackFrame(EXC_RETURN_Type excReturn, bit fourByteAlign)
    // Calculate the size of the integer part of the stack frame
    toSecure = HaveSecurityExt() && excReturn.S == '1';
    if toSecure && (excReturn.ES == '0' ||
                    excReturn.DCRS == '0') then
        framesize = 0x48;
    else
        framesize = 0x20;
    // Add on the size of the FP part of the stack frame if present
    if HaveMveOrFPExt() && excReturn.FType == '0' then
        if toSecure && FPCCR_S.TS == '1' then
            framesize = framesize + 0x88;
        else
            framesize = framesize + 0x48;

    // Update stack pointer. NOTE: Stack pointer limit not checked on exception
    // return as stack pointer guaranteed to be ascending not descending.
    mode   = if excReturn.Mode == '1' then PEMode_Thread else PEMode_Handler;
    spName = LookUpSP_with_security_mode(toSecure, mode);
    exc    = _SP(spName, FALSE, TRUE, (_SP(spName) + framesize) OR ZeroExtend(fourByteAlign:'
        00',32));
    assert exc.fault == NoFault;
```

### E2.1.54 ConsumptionOfSpeculativeDataBarrier

```
// Consumption of Speculative Data Barrier
// =======================================
// Perform a Consumption of Speculative Data Barrier operation.

ConsumptionOfSpeculativeDataBarrier();
```

### E2.1.55 Coproc_Accepted

```
// Coproc_Accepted
// ===============
// Check whether a coprocessor accepts an instruction.

boolean Coproc_Accepted(integer cp_num, bits(32) instr);
```

### E2.1.56 Coproc_DoneLoading

```
// Coproc_DoneLoading
// ==================
// Check whether enough 32-bit words have been loaded for an LDC instruction

boolean Coproc_DoneLoading(integer cp_num, bits(32) instr);
```

### E2.1.57 Coproc_DoneStoring

```
// Coproc_DoneStoring
// ==================
// Check whether enough 32-bit words have been stored for a STC instruction

boolean Coproc_DoneStoring(integer cp_num, bits(32) instr);
```

### E2.1.58 Coproc_GetOneWord

```
// Coproc_GetOneWord
// =================
// Gets the 32-bit word for an MRC instruction from the coprocessor
```

```
4
5  bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr);
```

### E2.1.59 Coproc_GetTwoWords

```
1  // Coproc_GetTwoWords
2  // ==================
3  // Get two 32-bit words for an MRRC instruction from the coprocessor
4
5  (bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr);
```

### E2.1.60 Coproc_GetWordToStore

```
1  // Coproc_GetWordToStore
2  // =====================
3  // Gets the next 32-bit word to store for an STC instruction from the coprocessor
4
5  bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr);
```

### E2.1.61 Coproc_InternalOperation

```
1  // Coproc_InternalOperation
2  // ========================
3  // Instructs a coprocessor to perform the internal operation requested
4  // by a CDP instruction
5
6  Coproc_InternalOperation(integer cp_num, bits(32) instr);
```

### E2.1.62 Coproc_SendLoadedWord

```
1  // Coproc_SendLoadedWord
2  // =====================
3  // Sends a loaded 32-bit word for an LDC instruction to the coprocessor
4
5  Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr);
```

### E2.1.63 Coproc_SendOneWord

```
1  // Coproc_SendOneWord
2  // ==================
3  // Sends the 32-bit word for an MCR instruction to the coprocessor
4
5  Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr);
```

### E2.1.64 Coproc_SendTwoWords

```
1  // Coproc_SendTwoWords
2  // ===================
3  // Send two 32-bit words for an MCRR instruction to the coprocessor.
4
5  Coproc_SendTwoWords(bits(32) word2, bits(32) word1, integer cp_num, bits(32) instr);
```

### E2.1.65 countLeadingSignBits

```
1  // CountLeadingSignBits()
2  // ======================
3
4  integer CountLeadingSignBits(bits(N) x)
5      return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

### E2.1.66 countLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - 1 - HighestSetBit(x);
```

### E2.1.67 CreateException

```
// CreateException()
// =================

ExcInfo CreateException(integer exception, boolean forceSecurity,
                        boolean isSecure, boolean isSynchronous)

    // Work out the effective target state of the exception
    if HaveSecurityExt() then
        if !forceSecurity then
            isSecure = ExceptionTargetsSecure(exception, isSecure);
    else
        isSecure = FALSE;

    // An implementation without Security Extensions cannot cause a fault targetting
    // Secure state
    assert HaveSecurityExt() || !isSecure;

    // Get the remaining exception details
    (escalateToHf, termInst) = ExceptionDetails(exception, isSecure, isSynchronous);

    // Fill in the default exception info
    info                  = DefaultExcInfo();
    info.fault            = exception;
    info.termInst         = termInst;
    info.origFault        = exception;
    info.origFaultIsSecure = isSecure;

    // Check for HardFault escalation
    // NOTE: In same cases (for example faults during lazy floating-point state preservation)
    //       the decision to escalate below is ignored and instead based on the info.
        origFault*
    //       fields and other factors.
    if escalateToHf && info.fault != HardFault then
        // Update the exception info with the escalation details, including
        // whether there's a change in destination Security state.
        info.fault        = HardFault;
        isSecure          = ExceptionTargetsSecure(HardFault, isSecure);
        (escalateToHf, -) = ExceptionDetails(HardFault, isSecure, isSynchronous);

    // If the requested exception was already a HardFault then we can't escalate
    // to a HardFault, so lockup. NOTE: Asynchronous BusFaults never cause
    // lockups, if the BusFault is disabled it escalates to a HardFault that is
    // pended.
    if escalateToHf && isSynchronous && info.fault == HardFault then
        info.lockup = TRUE;

    // Fill in the remaining exception info
    info.isSecure = isSecure;
    return info;

ExcInfo CreateException(integer exception, boolean forceSecurity, boolean isSecure)
    return CreateException(exception, forceSecurity, isSecure, TRUE);

ExcInfo CreateException(integer exception)
    return CreateException(exception, FALSE, IsSecure(), TRUE);
```

### E2.1.68 CurrentCond

```
// CurrentCond()
// =============
// Returns condition specifier of current instruction.

bits(4) CurrentCond();
```

### E2.1.69 CurrentMode

```
// CurrentMode()
// =============

PEMode CurrentMode()
    return if UInt(IPSR) == NoFault then PEMode_Thread else PEMode_Handler;
```

### E2.1.70 CurrentModeIsPrivileged

```
// CurrentModeIsPrivileged()
// =========================

boolean CurrentModeIsPrivileged()
    return CurrentModeIsPrivileged(IsSecure());

boolean CurrentModeIsPrivileged(boolean isSecure)
    nPriv = if isSecure then CONTROL_S.nPRIV else CONTROL_NS.nPRIV;
    return (CurrentMode() == PEMode_Handler || nPriv == '0');
```

### E2.1.71 D

```
// D[]
// ===

// Non-assignment form

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    return _S[(n*2)+1]:_S[n*2];

// Assignment form

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    _S[(n*2)+1] = value<63:32>;
    _S[n*2]     = value<31:0>;
    return;
```

### E2.1.72 DAPCheck

```
// DAPCheck()
// ==========

(boolean, boolean, boolean) DAPCheck(bits(32) address, boolean isPriv,
    boolean isSecure, boolean isWrite)

    assert(HaveSecurityExt() || !isSecure);
    err = FALSE;

    // DAP access falls back to Nonsecure when secure debug disabled
    isSecure = isSecure && DHCSR.S_SDE == '1';

    // DAP access are demoted to unprivileged when unprivileged only debug is enabled
    if (isSecure && DHCSR.S_SUIDE == '1') || (!isSecure && DHCSR.S_NSUIDE == '1') then
```

```
        isPriv = FALSE;

    if !(HaltingDebugAllowed()                 ||
         (isSecure  && DHCSR.S_SUIDE  == '1') ||
         (!isSecure && DHCSR.S_NSUIDE == '1')) then

        // Allow authorized unprivileged DAP requests.
        if DAUTHCTRL_S.UIDAPEN  == '1'|| DAUTHCTRL_NS.UIDAPEN == '1' then
            err    = FALSE;
            isPriv = FALSE;

        // Otherwise handle accesses based on NonInvasiveDebugAllowed or
        // region-specific rules
        else
            // Accesses are denied, except where explicitly allowed below
            err = TRUE;
            case address of
                when '1110 0000 0000 xxxx  xxxx 1111 1011 0xxx'
                    err = !NoninvasiveDebugAllowed();        // CoreSight software lock
                when '1110 0000 0000 xxxx  xxxx 1111 1101 xxxx'
                    err = isWrite;                           // All ID registers RO
                when '1110 0000 0000 xxxx  xxxx 1111 111x xxxx'
                    err = isWrite;                           // All ID registers RO
                when '1110 0000 0000 0000  0000 xxxx xxxx xxxx'
                    err = !NoninvasiveDebugAllowed();        // ITM
                when '1110 0000 0000 0000  0001 xxxx xxxx xxxx'
                    err = !NoninvasiveDebugAllowed();        // DWT
                when '1110 0000 0000 0000  0011 xxxx xxxx xxxx'
                    err = !NoninvasiveDebugAllowed();        // PMU
                when '1110 0000 0000 0100  0000 xxxx xxxx xxxx'
                    err = FALSE;                             // TPIU
                when '1110 0000 0000 0100  0001 xxxx xxxx xxxx'
                    err = FALSE;                             // ETM
                when '1110 0000 0000 1111  1111 xxxx xxxx xxxx'
                    err = isWrite;                           // ROM Table
                otherwise
                    address_uint = UInt(address);
                    if address_uint >= 0xE0042000 && address_uint <= 0xE00FEFFF then
                        err = boolean IMPLEMENTATION_DEFINED "IMPDEF DAP region";
                    elsif address_uint >= 0xE0100000 then
                        err = !NoninvasiveDebugAllowed();    // Vendor Sys

    return (isSecure, isPriv, err);
```

### E2.1.73 DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================
// Perform a Data Memory Barrier operation

DataMemoryBarrier(bits(4) option);
```

### E2.1.74 DataSynchronizationBarrier

```
// DataSynchronizationBarrier
// ==========================
// Perform a data synchronization barrier operation

DataSynchronizationBarrier(bits(4) option);
```

### E2.1.75 DeActivate

```
// DeActivate()
// ============

```

```
DeActivate(integer returningExceptionNumber, boolean targetDomainSecure)
    // To prevent the execution priority remaining negative (and therefore
    // masking HardFault) when returning from NMI / HardFault with a corrupted
    // IPSR value, the active bits corresponding to the execution priority are
    // cleared if the raw execution priority (ie the priority before FAULTMASK
    // and other priority boosting is considered) is negative.
    rawPri = RawExecutionPriority();
    if rawPri == -1 then
        SetActive(HardFault, AIRCR.BFHFNMINS == '0', FALSE);
    elsif rawPri == -2 then
        SetActive(NMI,       AIRCR.BFHFNMINS == '0', FALSE);
    elsif rawPri == -3 then
        SetActive(HardFault, TRUE,                   FALSE);
    else
        secure = HaveSecurityExt() && targetDomainSecure;
        SetActive(returningExceptionNumber, secure, FALSE);

    /* PRIMASK and BASEPRI unchanged on exception exit */
    if HaveMainExt() && rawPri >= 0 then
        // clear FAULTMASK for exception security domain on any return except
        // NMI and HardFault
        if HaveSecurityExt() && targetDomainSecure then
            FAULTMASK_S<0> = '0';
        else
            FAULTMASK_NS<0> = '0';
    return;
```

### E2.1.76 Debug_authentication

```
// In the recommended CoreSight interface, there are four signals for external debug
// authentication, DBGEN, SPIDEN, NIDEN and SPNIDEN. Each signal is active-HIGH.

signal DBGEN;
signal SPIDEN;
signal NIDEN;
signal SPNIDEN;
```

### E2.1.77 DebugCanMaskInts

```
// DebugCanMaskInts()
// ==================

boolean DebugCanMaskInts(boolean secure)
    return (HaltingDebugAllowed() && DHCSR.C_DEBUGEN == '1' &&
            (!secure || DHCSR.S_SDE == '1') && DHCSR.C_MASKINTS == '1');
```

### E2.1.78 DebugRegisterTransfer

```
// DebugRegisterTransfer()
// =======================

DebugRegisterTransfer(bits(7) reg, boolean isWrite)
    unprivDbgS  = HaveUDE() && DHCSR.S_SUIDE  == '1';
    unprivDbgNS = HaveUDE() && DHCSR.S_NSUIDE == '1';
    unprivDbg   = if IsSecure() then unprivDbgS else unprivDbgNS;

    if ((UInt(reg) >= UInt(DCRSR_REGSEL_R_LOW) && UInt(reg) <= UInt(DCRSR_REGSEL_R_HIGH)) ||
        reg == DCRSR_REGSEL_LR) then
        if isWrite then
            R[UInt(reg)] = DCRDR;
        else
            DCRDR = R[UInt(reg)];

    elsif reg == DCRSR_REGSEL_SP then
        if isWrite then
```

```
            // This requires skipping stack limit checking, hence a direct _RName access is used
            _RName[LookUpRName(UInt(reg))] = DCRDR<31:2>:'00';
        else
            DCRDR = _RName[LookUpRName(UInt(reg))];

    elsif reg == DCRSR_REGSEL_DBGRETADDR then
        if isWrite then
            BranchTo(DCRDR, TRUE);
        else
            DCRDR = _RName[RNamesPC];

    elsif reg == DCRSR_REGSEL_XPSR then
        if isWrite then
            if !unprivDbg then
                XPSR = DCRDR<31:0>;
            else
                EAPSR = DCRDR<31:0>;
        else
            if !unprivDbg then
                DCRDR<31:0> = XPSR;
            else
                DCRDR<31:0> = EAPSR;

    elsif reg == DCRSR_REGSEL_SP_MAIN then
        if isWrite then
            if !unprivDbg then
                SP_Main = DCRDR;
        else
            if !unprivDbg then
                DCRDR = SP_Main;
            else
                DCRDR = Zeros();

    elsif reg == DCRSR_REGSEL_SP_PROCESS then
        if isWrite then
            SP_Process = DCRDR;
        else
            DCRDR = SP_Process;

    elsif reg == DCRSR_REGSEL_STATE then
        if isWrite then
            if !unprivDbg then
                CONTROL<7:0> = DCRDR<31:24>;
                if HaveMainExt() then
                    FAULTMASK<7:0> = DCRDR<23:16>;
                    BASEPRI<7:0>   = DCRDR<15:8>;
                PRIMASK<7:0> = DCRDR<7:0>;
            else
                if DHCSR.S_SDE == '1' then
                    CONTROL.SFPA = DCRDR<27>;
                CONTROL.FPCA = DCRDR<26>;
        else
            if !unprivDbg then
                DCRDR<31:24> = CONTROL<7:0>;
                if HaveMainExt() then
                    DCRDR<23:16> = FAULTMASK<7:0>;
                    DCRDR<15:8>  = BASEPRI<7:0>;
                else
                    DCRDR<23:8> = Zeros(16);
                DCRDR<7:0> = PRIMASK<7:0>;
            else
                DCRDR<31:0> = Zeros(4) : (CONTROL.SFPA AND DHCSR.S_SDE) : CONTROL.FPCA : Zeros(26);

    elsif reg == DCRSR_REGSEL_MSP_NS && HaveSecurityExt() then
        if isWrite then
            // Unprivileged-only debug is restricted even if MSP is being used by
            // unprivileged execution, a safe restriction that removes the
```

```
            // requirement to check other conditions here.
            if !unprivDbgNS then
                SP_Main_NonSecure = DCRDR;
        else
            if !unprivDbgNS then
                DCRDR = SP_Main_NonSecure;
            else
                DCRDR<31:0> = Zeros();

    elsif reg == DCRSR_REGSEL_PSP_NS && HaveSecurityExt() then
        if isWrite then
            SP_Process_NonSecure = DCRDR;
        else
            DCRDR = SP_Process_NonSecure;

    elsif reg == DCRSR_REGSEL_MSP_S && HaveSecurityExt() then
        if isWrite then
            // Unprivileged-only debug is restricted even if MSP is being used by
            // unprivileged execution, a safe restriction that removes the
            // requirement to check other conditions here.
            if DHCSR.S_SDE == '1' && !unprivDbgS then
                SP_Main_Secure = DCRDR;
        else
            if DHCSR.S_SDE == '1' && !unprivDbgS then
                DCRDR = SP_Main_Secure;
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_PSP_S && HaveSecurityExt() then
        if isWrite then
            if DHCSR.S_SDE == '1' then
                SP_Process_Secure = DCRDR;
        else
            if DHCSR.S_SDE == '1' then
                DCRDR = SP_Process_Secure;
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_MSPLIM_S && HaveSecurityExt() then
        if isWrite then
            if DHCSR.S_SDE == '1' && !unprivDbgS then
                MSPLIM_S = DCRDR<31:0>;
        else
            if DHCSR.S_SDE == '1' && !unprivDbgS then
                DCRDR<31:0> = MSPLIM_S;
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_PSPLIM_S && HaveSecurityExt() then
        if isWrite then
            if DHCSR.S_SDE == '1' then
                PSPLIM_S = DCRDR<31:0>;
        else
            if DHCSR.S_SDE == '1' then
                DCRDR<31:0> = PSPLIM_S;
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_MSPLIM_NS && HaveMainExt() then
        if isWrite then
            if !unprivDbgNS then
                MSPLIM_NS = DCRDR<31:0>;
        else
            if !unprivDbgNS then
                DCRDR<31:0> = MSPLIM_NS;
            else
                DCRDR<31:0> = Zeros();

    elsif reg == DCRSR_REGSEL_PSPLIM_NS && HaveMainExt() then
```

```
        if isWrite then
            PSPLIM_NS = DCRDR<31:0>;
        else
            DCRDR<31:0> = PSPLIM_NS;

    elsif reg == DCRSR_REGSEL_FPSCR && (HaveFPExt() || HaveMve()) then
        if isWrite then
            if CanDebugAccessFP() then
                FPSCR = DCRDR<31:0>;
        else
            if CanDebugAccessFP() then
                DCRDR<31:0> = FPSCR;
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_STATE_S && HaveSecurityExt() then
        if isWrite then
            if DHCSR.S_SDE == '1' then
                if !unprivDbgS then
                    CONTROL_S<7:0> = DCRDR<31:24>;
                    if HaveMainExt() then
                        FAULTMASK_S<7:0> = DCRDR<23:16>;
                        BASEPRI_S<7:0>   = DCRDR<15:8>;
                    PRIMASK_S<7:0> = DCRDR<7:0>;
                else
                    CONTROL_S.SFPA = DCRDR<27>;
                    CONTROL_S.FPCA = DCRDR<26>;
        else
            if DHCSR.S_SDE == '1' then
                if !unprivDbgS then
                    DCRDR<31:24> = CONTROL_S<7:0>;
                    if HaveMainExt() then
                        DCRDR<23:16> = FAULTMASK_S<7:0>;
                        DCRDR<15:8>  = BASEPRI_S<7:0>;
                    else
                        DCRDR<23:8>  = Zeros(16);
                    DCRDR<7:0>  = PRIMASK_S<7:0>;
                else
                    DCRDR = Zeros(4) : CONTROL_S.SFPA : CONTROL_S.FPCA : Zeros(26);
            else
                DCRDR = Zeros(32);

    elsif reg == DCRSR_REGSEL_STATE_NS && HaveSecurityExt() then
        if isWrite then
            if !unprivDbgNS then
                CONTROL_NS<7:0> = DCRDR<31:24>;
                if HaveMainExt() then
                    FAULTMASK_NS<7:0> = DCRDR<23:16>;
                    BASEPRI_NS<7:0>   = DCRDR<15:8>;
                PRIMASK_NS<7:0> = DCRDR<7:0>;
            else
                CONTROL_NS.FPCA = DCRDR<26>;
        else
            if !unprivDbgNS then
                DCRDR<31:24> = CONTROL_NS<7:0>;
                if HaveMainExt() then
                    DCRDR<23:16> = FAULTMASK_NS<7:0>;
                    DCRDR<15:8>  = BASEPRI_NS<7:0>;
                else
                    DCRDR<23:8>  = Zeros(16);
                DCRDR<7:0> = PRIMASK_NS<7:0>;
            else
                DCRDR = Zeros(5) : CONTROL_NS.FPCA : Zeros(26);

    elsif reg == DCRSR_REGSEL_VPR && HaveMve() then
        if isWrite then
            if CanDebugAccessFP() then
                VPR = DCRDR<31:0>;
        else
```

```
            if CanDebugAccessFP() then
                DCRDR = VPR<31:0>;
            else
                DCRDR = Zeros(32);

    elsif UInt(reg) >= UInt(DCRSR_REGSEL_S_LOW) && UInt(reg) <= UInt(DCRSR_REGSEL_S_HIGH) &&
         (HaveFPExt() || HaveMve()) then
        if isWrite then
            if CanDebugAccessFP() then
                _S[UInt(reg<5:0>)] = DCRDR;
        else
            if CanDebugAccessFP() then
                DCRDR = _S[UInt(reg<5:0>)];
            else
                DCRDR = Zeros(32);

    else
        DCRDR = bits(32) UNKNOWN;
```

### E2.1.79 DecodeExecute

```
// DecodeExecute
// =============
// Decode instruction and execute

DecodeExecute(bits(32) instr, bits(32) pc, boolean isT16, bits(4) defaultCond);
```

### E2.1.80 DecodeImmShift

```
// DecodeImmShift()
// ================

(SRType, integer) DecodeImmShift(bits(2) sr_type, bits(5) imm5)

    case sr_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);
```

### E2.1.81 DecodeRegShift

```
// DecodeRegShift()
// ================

SRType DecodeRegShift(bits(2) sr_type)
    case sr_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;
```

### E2.1.82 DefaultCond

```
// DefaultCond()
// =============

bits(4) DefaultCond()
    // If in an IT block us the IT condition, otherwise set the condition to
    // always (I.E. 0xE).
    // NOTE: This is only the default condition, as it may be overridden by an
    //       explicit condition code in the instruction itself.
    if ITSTATE<3:0> == Zeros(4) then
        cond = 0xE<3:0>;
    else
        cond = ITSTATE<7:4>;
    return cond;
```

### E2.1.83 DefaultExcInfo

```
// DefaultExcInfo()
// ================

ExcInfo DefaultExcInfo()
    ExcInfo exc;

    exc.fault      = NoFault;
    exc.origFault  = NoFault;
    exc.isSecure   = boolean UNKNOWN;
    exc.isTerminal = FALSE;
    exc.inExcTaken = FALSE;
    exc.lockup     = FALSE;
    exc.termInst   = TRUE;
    return exc;
```

### E2.1.84 DefaultMemoryAttributes

```
// DefaultMemoryAttributes()
// =========================

MemoryAttributes DefaultMemoryAttributes(bits(32) address)

    MemoryAttributes mematttrs;

    case address<31:29> of
        when '000'
            mematttrs.memtype = MemType_Normal;
            mematttrs.device = DeviceType UNKNOWN;
            mematttrs.innerattrs = '10';
            mematttrs.shareable = FALSE;
        when '001'
            mematttrs.memtype = MemType_Normal;
            mematttrs.device = DeviceType UNKNOWN;
            mematttrs.innerattrs = '01';
            mematttrs.shareable = FALSE;
        when '010'
            mematttrs.memtype = MemType_Device;
            mematttrs.device = DeviceType_nGnRE;
            mematttrs.innerattrs = '00';
            mematttrs.shareable = TRUE;
        when '011'
            mematttrs.memtype = MemType_Normal;
            mematttrs.device = DeviceType UNKNOWN;
            mematttrs.innerattrs = '01';
            mematttrs.shareable = FALSE;
        when '100'
            mematttrs.memtype = MemType_Normal;
            mematttrs.device = DeviceType UNKNOWN;
            mematttrs.innerattrs = '10';
            mematttrs.shareable = FALSE;
        when '101'
```

```
            mematttrs.memtype = MemType_Device;
            mematttrs.device = DeviceType_nGnRE;
            mematttrs.innerattrs = '00';
            mematttrs.shareable = TRUE;
        when '110'
            mematttrs.memtype = MemType_Device;
            mematttrs.device = DeviceType_nGnRE;
            mematttrs.innerattrs = '00';
            mematttrs.shareable = TRUE;
        when '111'
            if address<28:20> == '000000000' then
                mematttrs.memtype = MemType_Device;
                mematttrs.device = DeviceType_nGnRnE;
                mematttrs.innerattrs = '00';
                mematttrs.shareable = TRUE;
            else
                mematttrs.memtype = MemType_Device;
                mematttrs.device = DeviceType_nGnRE;
                mematttrs.innerattrs = '00';
                mematttrs.shareable = TRUE;

    // Outer attributes are the same as the inner attributes in all cases.
    mematttrs.outerattrs = mematttrs.innerattrs;
    mematttrs.outershareable = mematttrs.shareable;

    // Setting as UNKNOWN by default. This flag will be  overwritten based on
    // SAU/IDAU checking in SecurityCheck()
    mematttrs.NS = boolean UNKNOWN;
    return mematttrs;
```

## E2.1.85 DefaultPermissions

```
// DefaultPermissions()
// ====================

Permissions DefaultPermissions(bits(32) address)

    Permissions perms;

    perms.ap          = '01';
    perms.apValid     = TRUE;
    perms.region      = Zeros(8);
    perms.regionValid = FALSE;

    case address<31:29> of
        when '000'
            perms.xn = '0';
        when '001'
            perms.xn = '0';
        when '010'
            perms.xn = '1';
        when '011'
            perms.xn = '0';
        when '100'
            perms.xn = '0';
        when '101'
            perms.xn = '1';
        when '110'
            perms.xn = '1';
        when '111'
            perms.xn = '1';

    return perms;
```

## E2.1.86 DerivedLateArrival

```
// DerivedLateArrival()
```

```
// ====================

DerivedLateArrival(integer pePriority, integer peNumber, boolean peIsSecure, ExcInfo deInfo,
                   integer oeNumber, boolean oeIsSecure, EXC_RETURN_Type excReturn)
    // PE: the pre-empted exception - before exception entry
    // OE: the original exception - exception entry
    // DE: the derived exception - fault on exception entry

    // Get the priorities of the exceptions
    // xePriority: the lower the value, the higher the priority
    oePriority = ExceptionPriority(oeNumber, oeIsSecure, FALSE);
    // NOTE: Comparison of dePriority against PE priority and possible
    // escalation to HardFault has already occurred. See CreateException().

    // Is the derived exception a DebugMonitor
    if HaveMainExt() then
        deIsDbgMonFault = (deInfo.origFault == DebugMonitor);
    else
        deIsDbgMonFault = FALSE;

    // Work out which fault to take, and what the target domain is
    if deInfo.isTerminal then
        // Derived exception is terminal and prevents the original exception
        // being taken (eg fault on vector fetch). As a result the derived
        // exception is treated as a HardFault.
        targetIsSecure = deInfo.isSecure;
        targetFault    = deInfo.fault;
        // If the derived fault does not have sufficient priority to pre-empt
        // lockup instead of taking it.
        if !ComparePriorities(deInfo, FALSE, oePriority, oeNumber, oeIsSecure) then
            _ = ExceptionTaken(oeNumber, deInfo.inExcTaken, oeIsSecure, IgnoreFaults_ALL,
                excReturn);
            // Since execution of original exception cannot be started, lockup
            // at the current priority level. That is the priority of the original
            // exception.
            Lockup(TRUE);
    elsif deIsDbgMonFault && !ComparePriorities(deInfo, TRUE, pePriority, peNumber,
        peIsSecure) then
        // Ignore the DebugMonitorFault and take original exception
        SetPending(DebugMonitor, deInfo.isSecure, FALSE);
        targetFault    = oeNumber;
        targetIsSecure = oeIsSecure;
    elsif ComparePriorities(deInfo, FALSE, oePriority, oeNumber, oeIsSecure) then
        // Derive exception has a higher priority (that is a lower value) than the
        // original exception, so the derived exception first. Tail-chaining
        // IMPLEMENTATION DEFINED
        targetFault    = deInfo.fault;
        targetIsSecure = deInfo.isSecure;
    else
        // If the derived exception caused a lockup then this must be handled
        // now as the lockup cannot be pended until the original exception
        // returns
        if deInfo.lockup then
            // Lockup at the priority of the original exception being entered.
            _ = ExceptionTaken(oeNumber, deInfo.inExcTaken, oeIsSecure, IgnoreFaults_ALL,
                excReturn);
            Lockup(TRUE);
        else
            // DE will be pended below, start execution of the OE
            targetFault    = oeNumber;
            targetIsSecure = oeIsSecure;

    // If none of the tests above have triggered a lockup (which would have
    // terminated execution of the pseudocode) then the derived exception
    // must be pended and any escalation syndrome info generated
    if HaveMainExt() &&
       (deInfo.fault     == HardFault) &&
       (deInfo.origFault != HardFault) then
        HFSR.FORCED = '1';
```

```
    SetPending(deInfo.fault, deInfo.isSecure, TRUE);

    // Take the target exception. NOTE: None terminal faults are ignored when
    // handling the derived exception, allowing forward progress to be made.
    (excInfo, excReturn) = ExceptionTaken(targetFault, deInfo.inExcTaken, targetIsSecure,
        IgnoreFaults_STACK, excReturn);
    // If trying to take the resulting exception results in another fault, then handle
    // the derived derived fault.
    if excInfo.fault != NoFault then
        DerivedLateArrival(pePriority, peNumber, peIsSecure, excInfo, targetFault,
            targetIsSecure, excReturn);
```

### E2.1.87 DeviceType

```
// Types of memory

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE
    };
```

### E2.1.88 DWT_AddressCompare

```
// DWT_AddressCompare()
// ====================
// Returns a pair of values. The first result is whether the (masked) addresses are equal,
// where the access address (addr) is masked according to DWT_FUNCTION<n>.DATAVSIZE and the
// comparator address (compaddr) is masked according to the access size. The second result
// is whether the (unmasked) addr is greater than the (unmasked) compaddr.

(boolean,boolean) DWT_AddressCompare(bits(32) addr, bits(32) compaddr, integer size,
                                     integer compsize)
    // addr must be a multiple of size. Unaligned accesses are split into smaller accesses.
    assert Align(addr, size) == addr;

    // compaddr must be a multiple of compsize
    if Align(compaddr, compsize) != compaddr then UNPREDICTABLE;

    addrmatch   = (Align(addr, compsize) == Align(compaddr, size));
    addrgreater = (UInt(addr) > UInt(compaddr));
    return (addrmatch,addrgreater);
```

### E2.1.89 DWT_CycCountMatch

```
// DWT_CycCountMatch
// =================
// Check for DWT cycle count match. This is called for each increment of
// DWT_CYCCNT.

DWT_CycCountMatch()
    boolean trigger_debug_event = FALSE;
    boolean debug_event = FALSE;
    N = UInt(DWT_CTRL.NUMCOMP);
    if N == 0 then return;              // No comparator support
    secure_match = IsSecure() && DWT_CTRL.CYCDISS == '1';
    for i = 0 to N-1
        if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
        if DWT_FUNCTION[i].MATCH == '0001' && DWT_ValidMatch(i, secure_match)
            && DWT_CYCCNT == DWT_COMP[i] then
            DWT_FUNCTION[i].MATCHED = '1';
            debug_event = DWT_FUNCTION[i].ACTION == '01';
        trigger_debug_event = trigger_debug_event || debug_event;

    // Setting the debug event if atleast one comparator matches
    if trigger_debug_event then
        debug_event = SetDWTDebugEvent(secure_match);
    return;
```

### E2.1.90 DWT_DataAddressMatch

```
// DWT_DataAddressMatch()
// ======================
// Check for match of access at "daddr". "dsize", "read" and "NSreq" are the attributes
// for the access. Note that for a load or store instruction, "NSreq" is the current
// Security state of the PE, but this is not necessarily true for a hardware stack
// push/pop or vector table access. "NSreq" might not be the same as the "NSattr"
// attribute the PE finally uses to make the access.
// If comparators 'm' and 'm+1' form an Data Address Range comparator, then this function
// returns the range match result when N=m+1.

boolean DWT_DataAddressMatch(integer N, bits(32) daddr, integer dsize, boolean read,
                             boolean NSreq)
    assert N < UInt(DWT_CTRL.NUMCOMP) && dsize IN {1,2,4} && Align(daddr, dsize) == daddr;

    valid_match = DWT_ValidMatch(N, !NSreq);
    valid_addr = DWT_FUNCTION[N].MATCH == 'x1xx';

    if valid_match && valid_addr then
        if N != UInt(DWT_CTRL.NUMCOMP)-1 then
            linked_to_addr = DWT_FUNCTION[N+1].MATCH == '0111';   // Data Address Limit
            linked_to_data = DWT_FUNCTION[N+1].MATCH == '1011';   // Linked Data Value
        else
            linked_to_addr = FALSE;  linked_to_data = FALSE;

        case DWT_FUNCTION[N].MATCH<1:0> of
            when '00'  match_lsc = TRUE;   linked = FALSE;
            when '01'  match_lsc = !read;  linked = FALSE;
            when '10'  match_lsc = read;   linked = FALSE;
            when '11'

                case DWT_FUNCTION[N-1].MATCH<1:0> of
                    when '00'  match_lsc = TRUE;   linked = TRUE;
                    when '01'  match_lsc = !read;  linked = TRUE;
                    when '10'  match_lsc = read;   linked = TRUE;

        if !linked_to_addr then
            vsize = 2^UInt(DWT_FUNCTION[N].DATAVSIZE);
            (match_eq,match_gt) = DWT_AddressCompare(daddr, DWT_COMP[N], dsize, vsize);

            if linked then
                valid_match = DWT_ValidMatch(N-1, !NSreq);
                (lower_eq,lower_gt) = DWT_AddressCompare(daddr, DWT_COMP[N-1], dsize, 1);
                match_addr = valid_match && (lower_eq || lower_gt) && !match_gt;
            else
                match_addr = match_eq;
        else
            match_addr = FALSE;

        match = match_addr && match_lsc;
    else
        match = FALSE;

    return match;
```

### E2.1.91 DWT_DataMatch

```
// DWT_DataMatch()
// ===============
// Perform variois Data match checks for DWT

DWT_DataMatch(bits(32) daddr, integer dsize, bits(32) dvalue, boolean read, boolean NSreq)

    boolean trigger_debug_event = FALSE;
    boolean debug_event = FALSE;

```

```
    if !HaveDWT() || IsZero(DWT_CTRL.NUMCOMP) then return;              // No comparator
        support

    for i = 0 to UInt(DWT_CTRL.NUMCOMP) - 1
        if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
        daddr_match  = DWT_DataAddressMatch(i, daddr, dsize, read, NSreq);
        dvalue_match = DWT_DataValueMatch(i, daddr, dvalue, dsize, read, NSreq);

        // Data Address and Data Address Limit
        if daddr_match && DWT_FUNCTION[i].MATCH == '01xx' then
            // Data Address
            if DWT_FUNCTION[i].MATCH != '0111' then
                DWT_FUNCTION[i].MATCHED = '1';
                debug_event = DWT_FUNCTION[i].ACTION == '01';

            // Data Address with Data Address Limit
            else
                //ith comparator
                DWT_FUNCTION[i].MATCHED = bit UNKNOWN;
                // (i-1)th comparator
                DWT_FUNCTION[i-1].MATCHED = '1';
                debug_event = DWT_FUNCTION[i-1].ACTION == '01';

        // Data Value and Linked Data Value
        if dvalue_match && DWT_FUNCTION[i].MATCH == '10xx' then
            // Data Value
            if DWT_FUNCTION[i].MATCH != '1011' then
                DWT_FUNCTION[i].MATCHED = '1';
                debug_event = DWT_FUNCTION[i].ACTION == '01';

            // For Linked Data Value, daddr_match will be TRUE for [i-1]
            else
                DWT_FUNCTION[i].MATCHED = '1';
                debug_event = DWT_FUNCTION[i].ACTION == '01';

        // Data Address with Value
        if daddr_match && DWT_FUNCTION[i].MATCH == '11xx' then
            DWT_FUNCTION[i].MATCHED = '1';
            // No debug_event generated in the case of Data Address with Value

        trigger_debug_event = trigger_debug_event || debug_event;

    // Setting the debug event if at least one comparator matches
    if trigger_debug_event then
        debug_event = SetDWTDebugEvent(!NSreq);

    return;
```

### E2.1.92 DWT_DataValueMatch

```
// DWT_DataValueMatch()
// ====================
// Check for match of access of "dvalue" at "daddr". "dsize", "read" and "NSreq"
// are the attributes for the access. Note that for a load or store instruction,
// "NSreq" is the current Security state of the PE, but this is not necessarily
// true for a hardware stack push/pop or vector table access. "NSreq" might not
// be the same as the "NSattr" attribute the PE finally uses to make the access.

boolean DWT_DataValueMatch(integer N, bits(32) daddr, bits(32) dvalue, integer dsize,
                           boolean read, boolean NSreq)
    assert N < UInt(DWT_CTRL.NUMCOMP) && dsize IN {1,2,4} && Align(daddr,dsize) == daddr;

    valid_match = DWT_ValidMatch(N, !NSreq);
    valid_data = DWT_FUNCTION[N].MATCH<3:2> == '10';

    if valid_match && valid_data then
        case DWT_FUNCTION[N].MATCH<1:0> of
            when '00'  match_lsc = TRUE;   linked = FALSE;
```

```
            when '01'  match_lsc = !read;  linked = FALSE;
            when '10'  match_lsc = read;   linked = FALSE;
            when '11'
                case DWT_FUNCTION[N-1].MATCH<1:0> of
                    when '00'  match_lsc = TRUE;   linked = TRUE;
                    when '01'  match_lsc = !read;  linked = TRUE;
                    when '10'  match_lsc = read;   linked = TRUE;

        vsize = 2^UInt(DWT_FUNCTION[N].DATAVSIZE);

        // Determine which bytes of dvalue to look at in the comparison.
        if linked then
            byte_mask = '0000'; // Filled in below if there is an address match
            if DWT_DataAddressMatch(N-1, daddr, dsize, read, NSreq) then
                case (vsize,dsize) of
                    when (1,1)  byte_mask<0> = '1';
                    when (1,2)  byte_mask<UInt(DWT_COMP[N-1]<0>)> = '1';
                    when (1,4)  byte_mask<UInt(DWT_COMP[N-1]<1:0>)> = '1';
                    when (2,2)  byte_mask<1:0> = '11';
                    when (2,4)
                        byte_mask<UInt(DWT_COMP[N-1]<1:0>)+1:UInt(DWT_COMP[N-1]<1:0>)> = '11'
                            ;
                    when (4,4)  byte_mask = '1111';
                    otherwise   byte_mask = '0000';   // vsize > dsize: no match
        else
            case dsize of
                when 1  byte_mask = '0001';
                when 2  byte_mask = '0011';
                when 4  byte_mask = '1111';

        // Perform bitwise mask on the candidate data value
        bit_mask = (if HasArchVersion(Armv8p1) then DWT_VMASK[N] else Zeros(32));
        dvalue = (dvalue AND NOT bit_mask);

        // Split both values into byte lanes: DCBA and dcba.
        // This function relies on the values being correctly replicated across DWT_COMP[N].
        D = dvalue<31:24>; C = dvalue<23:16>; B = dvalue<15:8>; A = dvalue<7:0>;
        d = DWT_COMP[N]<31:24>; c = DWT_COMP[N]<23:16>;
        b = DWT_COMP[N]<15:8>;  a = DWT_COMP[N]<7:0>;

        // Partial results
        D_d = byte_mask<3> == '1' && D == d;
        C_c = byte_mask<2> == '1' && C == c;
        B_b = byte_mask<1> == '1' && B == b;
        A_a = byte_mask<0> == '1' && A == a;

        // Combined partial results
        BA_ba = B_b && A_a;
        DC_dc = D_d && C_c;
        DCBA_dcba = D_d && C_c && B_b && A_a;

        // Generate full results
        case (vsize,dsize) of
            when (1,-)          match_data = D_d || C_c || B_b || A_a;
            when (2,2), (2,4)   match_data = DC_dc || BA_ba;
            when (4,4)          match_data = DCBA_dcba;
            otherwise           match_data = FALSE;    // vsize > dsize: no match

        match = match_data && match_lsc;
    else
        match = FALSE;

    return match;
```

## E2.1.93 DWT_InstructionAddressMatch

```
// DWT_InstructionAddressMatch()
// =============================
```

```
// Check for match of instruction access at "Iaddr".
// If comparators 'm' and 'm+1' form an Instruction Address Range comparator, then this
// function returns the range match when N=m+1.

boolean DWT_InstructionAddressMatch(integer N, bits(32) Iaddr)
    assert N < UInt(DWT_CTRL.NUMCOMP) && Align(Iaddr, 2) == Iaddr;

    secure_match = IsSecure();
    valid_match = DWT_ValidMatch(N, secure_match);
    valid_instr = DWT_FUNCTION[N].MATCH == '001x';

    if valid_match && valid_instr then
        if N != UInt(DWT_CTRL.NUMCOMP)-1 then
            linked_to_instr = DWT_FUNCTION[N+1].MATCH == '0011';
        else
            linked_to_instr = FALSE;

        if DWT_FUNCTION[N].MATCH == '0011' then
            linked = TRUE;
        else
            linked = FALSE;

        if !linked_to_instr then
            (match_eq,match_gt) = DWT_AddressCompare(Iaddr, DWT_COMP[N], 2, 2);
            if linked then
                valid_match = DWT_ValidMatch(N-1, secure_match);
                (lower_eq,lower_gt) = DWT_AddressCompare(Iaddr, DWT_COMP[N-1], 2, 2);
                match_addr = valid_match && (lower_eq || lower_gt) && !match_gt;
            else
                match_addr = match_eq;
        else
            match_addr = FALSE;
        match = match_addr;
    else
        match = FALSE;

    return match;
```

## E2.1.94 DWT_InstructionMatch

```
// DWT_InstructionMatch()
// ======================
// Perform various Instruction Address checks for DWT

DWT_InstructionMatch(bits(32) Iaddr)

    boolean trigger_debug_event = FALSE;
    boolean debug_event = FALSE;

    if !HaveDWT() || IsZero(DWT_CTRL.NUMCOMP) then return;              // No comparator
        support

    for i = 0 to UInt(DWT_CTRL.NUMCOMP) - 1
        if IsDWTConfigUnpredictable(i) then UNPREDICTABLE;
        instr_addr_match = DWT_InstructionAddressMatch(i, Iaddr);
        if instr_addr_match then
            // Instruction Address
            if DWT_FUNCTION[i].MATCH == '0010' then
                DWT_FUNCTION[i].MATCHED = '1';
                debug_event = DWT_FUNCTION[i].ACTION == '01';

            // Instruction Address Limit
            elsif DWT_FUNCTION[i].MATCH == '0011' then
                DWT_FUNCTION[i].MATCHED = bit UNKNOWN;
                DWT_FUNCTION[i-1].MATCHED = '1';
                debug_event = DWT_FUNCTION[i-1].ACTION == '01';

            trigger_debug_event = trigger_debug_event || debug_event;
```

```
28
29      if trigger_debug_event then
30          debug_event = SetDWTDebugEvent(IsSecure());
31      return;
```

### E2.1.95 DWT_ValidMatch

```
1  // DWT_ValidMatch()
2  // ================
3  // Returns TRUE if this match is permitted by the current authentication controls, FALSE
       otherwise.
4
5  boolean DWT_ValidMatch(integer N, boolean secure_match)
6      if !HaveSecurityExt() then assert !secure_match;
7
8      // Check for disabled
9      if !NoninvasiveDebugAllowed() || DEMCR.TRCENA == '0' || DWT_FUNCTION[N].MATCH == '0000'
           then
10         return FALSE;
11
12     // Check for Debug event
13     if DWT_FUNCTION[N].ACTION == '01' then
14         hlt_en = CanHaltOnEvent(secure_match);
15         // Ignore priority when checking whether DebugMonitor activates DWT matches
16         mon_en = HaveDebugMonitor() && CanPendMonitorOnEvent(secure_match, FALSE, TRUE);
17         return (hlt_en || mon_en);
18     else
19         // Otherwise trace or trigger event
20         return !secure_match || SecureNoninvasiveDebugAllowed();
```

### E2.1.96 Elem

```
1  // Elem[]
2  // ======
3
4  // Non-assignment form
5
6  bits(size) Elem[bits(N) vector, integer e, integer size]
7      assert e >= 0 && (e+1)*size <= N;
8      return vector<(e+1)*size-1:e*size>;
9
10 bits(size) Elem[bits(N) vector, integer e]
11     return Elem[vector, e, size];
12
13 // Assignment form
14
15 Elem[bits(N) &vector, integer e, integer size] = bits(size) value
16     assert e >= 0 && (e+1)*size <= N;
17     vector<(e+1)*size-1:e*size> = value;
18     return;
19
20 Elem[bits(N) &vector, integer e] = bits(size) value
21     Elem[vector, e, size] = value;
22     return;
```

### E2.1.97 EndOfInstruction

```
1  // EndOfInstruction
2  // ================
3  // Terminates the processing of current instruction.
4
5  EndOfInstruction();
```

### E2.1.98 EventRegistered

```
// EventRegistered
// ===============
// Returns TRUE if PE Event Register is set to 1 and FALSE otherwise.

boolean EventRegistered();
```

### E2.1.99 ExceptionActiveBitCount

```
// ExceptionActiveBitCount()
// =========================

integer ExceptionActiveBitCount()
    integer count = 0;
    for i = 0 to MaxExceptionNum()
        for j = 0 to 1
            if IsActiveForState(i, j == 0) then
                count = count + 1;
    return count;
```

### E2.1.100 ExceptionDetails

```
// ExceptionDetails()
// ==================

(boolean, boolean) ExceptionDetails(integer exception, boolean isSecure, boolean
    isSynchronous)
    // Is the exception subject to escalation
    case exception of
        when HardFault
            termInst    = TRUE;
            canPend     = TRUE;
            canEscalate = TRUE;
        when MemManage
            termInst    = TRUE;
            if HaveMainExt() then
                val     = if isSecure then SHCSR_S else SHCSR_NS;
                canPend = val.MEMFAULTENA == '1';
            else
                canPend = FALSE;
            canEscalate = TRUE;
        when BusFault
            termInst    = isSynchronous;
            canPend     = if HaveMainExt()
                          then SHCSR_S.BUSFAULTENA == '1' else FALSE;
            // Async BusFaults only escalate if they are disabled
            canEscalate = termInst || !canPend;
        when UsageFault
            termInst    = TRUE;
            if HaveMainExt() then
                val     = if isSecure then SHCSR_S else SHCSR_NS;
                canPend = val.USGFAULTENA == '1';
            else
                canPend = FALSE;
            canEscalate = TRUE;
        when SecureFault
            termInst    = TRUE;
            canPend     = if HaveMainExt()
                          then SHCSR_S.SECUREFAULTENA == '1' else FALSE;
            canEscalate = TRUE;
        when SVCall
            termInst    = FALSE;
            canPend     = TRUE;
            canEscalate = TRUE;
        when DebugMonitor
            termInst    = TRUE;
            canPend     = HaveMainExt() && CanPendMonitorOnEvent(IsSecure(), TRUE, TRUE);
            canEscalate = TRUE;
```

```
        otherwise
            termInst    = FALSE;
            canEscalate = FALSE;

    // If the fault can escalate then check if exception can be taken immediately, or whether
    // it should escalate.
    // NOTE: In same cases (for example faults during lazy floating-point state preservation)
    //       the priority comparison below is ignored and the decision to escalate or not is
    //       based on other factors.
    escalateToHf = FALSE;
    if canEscalate then
        execPri = ExecutionPriority();
        excePri = ExceptionPriority(exception, isSecure, TRUE);
        if (excePri >= execPri) || !canPend then
            escalateToHf = TRUE;

    return (escalateToHf, termInst);
```

## E2.1.101 ExceptionEnabled

```
// ExceptionEnabled()
// ==================

boolean ExceptionEnabled(integer exception, boolean secure)
    assert 1 <= exception && exception < NUMEXN;
    if secure && !HaveSecurityExt() then
        enabled = FALSE;
    elsif exception < 16 then
        val = if secure then _SHCSR_S else _SHCSR_NS;
        case exception of
            when Reset
                enabled = TRUE;
            when NMI
                enabled = secure == (AIRCR_S.BFHFNMINS == '0');
            when HardFault
                enabled = secure || AIRCR_S.BFHFNMINS == '1';
            when MemManage
                enabled = val.MEMFAULTENA == '1';
            when BusFault
                enabled = ((_SHCSR_S.BUSFAULTENA == '1') &&
                           (secure == (AIRCR_S.BFHFNMINS == '0')));
            when UsageFault
                enabled = val.USGFAULTENA == '1';
            when SecureFault
                enabled = secure && _SHCSR_S.SECUREFAULTENA == '1';
            when SVCall
                enabled = TRUE;
            when DebugMonitor
                enabled = ((secure == (DEMCR.SDME == '1')) &&
                           !InstructionsInFlight());
            when PendSV
                enabled = !DebugCanMaskInts(secure);
            when SysTick
                enabled = ((!IsExceptionTargetConfigurable(SysTick) ||
                           (secure == (_ICSR_S.STTNS == '0'))) &&
                           !DebugCanMaskInts(secure));
            otherwise
                enabled = FALSE;
    else
        enabled = (IrqEnabled[exception-16] && ((NVIC_ITNS<exception-16> == '0') == secure)
            &&
                   !DebugCanMaskInts(secure));

    return enabled;
```

## E2.1.102 ExceptionEnabled

```
// ExceptionEnabled()
// ==================
// Checks whether the given exception is enabled.

boolean ExceptionEnabled(integer exception, boolean secure);
```

### E2.1.103 ExceptionEntry

```
// ExceptionEntry()
// ================
// Exception entry is modified according to the behavior of a derived
// exception, see DerivedLateArrival() also.

(ExcInfo, EXC_RETURN_Type) ExceptionEntry(integer exceptionType, boolean toSecure, boolean
    commitState)

    // PushStack() can abandon memory accesses if a fault occurs during the stacking
    // sequence.
    (exc, partialExcReturn) = PushStack(commitState);
    if exc.fault == NoFault then
        (exc, partialExcReturn) = ExceptionTaken(exceptionType, FALSE, toSecure,
            IgnoreFaults_NONE, partialExcReturn);
    return (exc, partialExcReturn);
```

### E2.1.104 ExceptionPriority

```
// ExceptionPriority()
// ===================

integer ExceptionPriority(integer n, boolean isSecure, boolean groupPri)
    if HaveMainExt() then
        assert 1 <= n && n < 512;
    else
        assert 1 <= n && n < 48;

    if n == Reset then                              // Reset
        result = -4;
    elsif n == NMI then                             // NMI
        result = -2;
    elsif n == HardFault then                       // HardFault
        if isSecure && AIRCR.BFHFNMINS == '1' then
            result = -3;
        else
            result = -1;
    elsif HaveMainExt() && n == MemManage then      // MemManage
        result = UInt(if isSecure then SHPR1_S.PRI_4 else SHPR1_NS.PRI_4);
    elsif HaveMainExt() && n == BusFault then       // BusFault
        result = UInt(SHPR1_S.PRI_5);
    elsif HaveMainExt() && n == UsageFault then     // UsageFault
        result = UInt(if isSecure then SHPR1_S.PRI_6 else SHPR1_NS.PRI_6);
    elsif HaveMainExt() && n == SecureFault then    // SecureFault
        result = UInt(SHPR1_S.PRI_7);
    elsif n == SVCall then                          // SVCall
        result = UInt(if isSecure then SHPR2_S.PRI_11 else SHPR2_NS.PRI_11);
    elsif HaveMainExt() && n == DebugMonitor then   // DebugMonitor
        result = UInt(SHPR3_S.PRI_12);
    elsif n == PendSV then                          // PendSV
        result = UInt(if isSecure then SHPR3_S.PRI_14 else SHPR3_NS.PRI_14);
    elsif n == SysTick                              // SysTick
          && ((HaveSysTick() == 2) ||
              (HaveSysTick() == 1  && ((_ICSR_S.STTNS == '0') == isSecure))) then
        result = UInt(if isSecure then SHPR3_S.PRI_15 else SHPR3_NS.PRI_15);
    elsif n >= 16 then                              // External interrupt (n-16)
        r = (n - 16) DIV 4;
        v = n MOD 4;
        result = UInt(NVIC_IPR[r]<v*8+7:v*8>);
    else                                            // Reserved exceptions
```

```
        result = 256;

    assert result IN {-4 .. 256};

    // Negative priorities (ie Reset, NMI, and HardFault) are not effected by
    // PRIGROUP or PRIS
    if result >= 0 then
        // Include the PRIGROUP effect
        if HaveMainExt() && groupPri then
            integer subgroupshift;
            if isSecure then
                subgroupshift = UInt(AIRCR_S.PRIGROUP);
            else
                subgroupshift = UInt(AIRCR_NS.PRIGROUP);
            integer groupvalue    = 2 << subgroupshift;
            integer subgroupvalue = result MOD groupvalue;
            result                = result - subgroupvalue;

        PriSNsPri = RestrictedNSPri();
        if (AIRCR_S.PRIS == '1') && !isSecure then
            result = (result >> 1) + PriSNsPri;

    assert result IN {-4 .. 256};
    return result;
```

### E2.1.105 ExceptionReturn

```
// ExceptionReturn()
// =================

(ExcInfo, EXC_RETURN_Type, boolean) ExceptionReturn(EXC_RETURN_Type excReturn)
    integer returningExceptionNumber = UInt(IPSR.Exception);

    (exc, excReturn) = ValidateExceptionReturn(excReturn, returningExceptionNumber);
    if exc.fault != NoFault then
        return (exc, excReturn, FALSE);

    if HaveSecurityExt() then
        excSecure   = excReturn.ES == '1';
        retToSecure = excReturn.S  == '1';
    else
        excSecure   = FALSE;
        retToSecure = FALSE;

    // Restore SPSEL for the Security state we are returning from.
    if excSecure then
        CONTROL_S.SPSEL  = excReturn.SPSEL;
    else
        CONTROL_NS.SPSEL = excReturn.SPSEL;

    returningExcIsSecure = excReturn.ES == '1';
    DeActivate(returningExceptionNumber, returningExcIsSecure);

    // If requested, clear the scratch FP values left in the caller saved
    // registers before returning/tail chaining.
    if HaveMveOrFPExt() && FPCCR.CLRONRET == '1' && CONTROL.FPCA == '1' then
        if FPCCR_S.LSPACT == '1' then
            SFSR.LSERR = '1';
            exc = CreateException(SecureFault);
            return (exc, excReturn, FALSE);
        else
            // Check if we have permission to clear the registers.
            if HasArchVersion(Armv8p1) then
                exc = CheckCPEnabled(10, TRUE, returningExcIsSecure);
                if exc.fault != NoFault then
                    return (exc, excReturn, FALSE);

            // Clear the FP / MVE registers
```

```
            InvalidateFPRegs(TRUE, FALSE);

    // If TailChaining is supported, check if there is a pending exception with
    // sufficient priority to be taken now. This check is done after the
    // previous exception is deactivated so the priority of the previous
    // exception doesn't mask any pending exceptions.
    // The position of TailChain() within this function is the earliest point
    // at which an tailchain is architecturally visible. Tail-chaining from a
    // later point is permissible.
    if boolean IMPLEMENTATION_DEFINED "Tail chaining supported" then
        (takeException, exception, excIsSecure) = PendingExceptionDetails();
        if takeException then
            (exc, excReturn) = TailChain(exception, excIsSecure, excReturn);
            return (exc, excReturn, TRUE);

    // Return to the background Security state
    if HaveSecurityExt() then
        CurrentState = if retToSecure
                       then SecurityState_Secure else SecurityState_NonSecure;

    // Sleep-on-exit performs equivalent behavior to the WFI instruction.
    // The position of SleepOnExit() within this function is the earliest point
    // at which it can be performed. Performing SleepOnExit from a later point
    // is permissible.
    if (excReturn.Mode == '1' && SCR.SLEEPONEXIT == '1' &&
        ExceptionActiveBitCount() == 0) then
            SleepOnExit();                          // IMPLEMENTATION DEFINED

    // Pop the stack and raise any exceptions that are generated
    exc = PopStack(excReturn);
    if exc.fault == NoFault then
        ClearExclusiveLocal(ProcessorID());
        SetEventRegister();                         // See WFE instruction for more details
        InstructionSynchronizationBarrier('1111');

    return (exc, excReturn, FALSE);
```

## E2.1.106 ExceptionTaken

```
// ExceptionTaken()
// ================

(ExcInfo, EXC_RETURN_Type) ExceptionTaken(integer exceptionNumber, boolean doTailChain,
                       boolean excIsSecure, IgnoreFaultsType ignoreFaults,
                       EXC_RETURN_Type excReturn)
    assert(HaveSecurityExt() || !excIsSecure);

    // If the background code was running in the Secure state that are some
    // additional steps that might need to be taken to protect the callee saved
    // registers
    exc = DefaultExcInfo();
    if HaveSecurityExt() && excReturn.S == '1' then
        if excIsSecure then         // Transitioning to Secure
            // If tail chaining is from Non-secure to Secure, then the callee registers
            // are already on stack. Set excReturn.DCRS accordingly
            if doTailChain && excReturn.ES == '0' then
                excReturn.DCRS = '0';
        else                        // Transitioning to Non-secure
            // If the callee registers aren't already on the stack push them now
            if excReturn.DCRS == '1' && !(doTailChain && excReturn.ES == '0') then
                exc = PushCalleeStack(doTailChain, excReturn);
            // Going to Non-secure exception. Set excReturn.DCRS to default
            // value
            excReturn.DCRS = '1';

    // Finalise excReturn value
    if excIsSecure then
        excReturn.SPSEL = CONTROL_S.SPSEL;
```

```
        excReturn.ES = '1';
    else
        excReturn.SPSEL = CONTROL_NS.SPSEL;
        excReturn.ES = '0';
    LR = excReturn;

    // Register clearing
    // Caller saved registers: These registers are cleared if exception targets
    // the Non-secure state, otherwise they are UNKNOWN. As of Armv8.1 the
    // registers are always cleared if the Security extension is implemented.
    // NOTE: The original values were pushed to the stack.
    if HaveSecurityExt() && (!excIsSecure || HasArchVersion(Armv8p1)) then
        callerRegValue = Zeros(32);
    else
        callerRegValue = bits(32) UNKNOWN;
    for n = 0 to 3
        R[n] = callerRegValue;
    R[12] = callerRegValue;
    EAPSR = callerRegValue;
    // Callee saved registers: If the background code was in the Secure state
    // these registers are cleared if the exception targets the Non-secure state,
    // and UNKNOWN if it targets the Secure state and the registers have been
    // pushed to the stack (as indicated by EXC_RETURN.DCRS).
    //
    // NOTE: Callee saved registers are preserved if the background code is
    //       Non-secure, or when the exception is Secure and the values have not
    //       been pushed to the stack.
    if HaveSecurityExt() && excReturn.S == '1' then
        if excIsSecure then
            if excReturn.DCRS == '0' then
                for n = 4 to 11
                    R[n] = bits(32) UNKNOWN;
        else
            for n = 4 to 11
                R[n] = Zeros();

    // If enabled, the IESB contains asynchronous RAS / BusFault errors to the background
    // context. This is conditional on there being no stacking faults -- if there are, the
    // errors will be synchronized when the subsequent exception is raised.
    if AIRCR.IESB == '1' then
        exc = MergeExcInfo(exc, SynchronizeBusFault());

    // If no errors so far (or errors that can be ignored) load the vector address
    if exc.fault == NoFault || ignoreFaults != IgnoreFaults_NONE then
        (exc, start) = Vector[exceptionNumber, excIsSecure];

    // The state or mode of processor is not updated if an exception is raised
    // during the entry sequence.
    if exc.fault == NoFault || ignoreFaults == IgnoreFaults_ALL then
        ActivateException(exceptionNumber, excIsSecure);
        SCS_UpdateStatusRegs();
        ClearExclusiveLocal(ProcessorID());
        SetEventRegister();                             // See WFE instruction for details
        InstructionSynchronizationBarrier('1111');
        // Start execution of handler
        EPSR.T = start<0>;
        // If EPSR.T == 0 then an exception is taken on the next
        // instruction:  UsageFault('Invalid State') if the Main Extension is
        // implemented; HardFault otherwise
        BranchTo(start<31:1>:'0');

    if exc.fault != NoFault then
        exc.inExcTaken = TRUE;

    return (exc, excReturn);
```

### E2.1.107 ExceptionTargetsSecure

```
// ExceptionTargetsSecure()
// ========================

// Determine the default Security state an exception is expected to target if the
// exception is not forced to a specific domain

boolean ExceptionTargetsSecure(integer exceptionNumber, boolean isSecure)
    if !HaveSecurityExt() then
        return FALSE;

    boolean targetSecure = FALSE;
    case exceptionNumber of
        when NMI
            targetSecure = AIRCR.BFHFNMINS == '0';

        when HardFault
            targetSecure = AIRCR.BFHFNMINS == '0' || isSecure;

        when MemManage
            targetSecure = isSecure;

        when BusFault
            targetSecure = AIRCR.BFHFNMINS == '0';

        when UsageFault
            targetSecure = isSecure;

        when SecureFault
            // SecureFault always targets Secure state
            targetSecure = TRUE;

        when SVCall
            targetSecure = isSecure;

        when DebugMonitor
            targetSecure = DEMCR.SDME == '1';

        when PendSV
            targetSecure = isSecure;

        when SysTick
            if HaveSysTick() == 2 then
                // If there is a SysTick for each domain, then the exception
                // targets the domain associated with the SysTick instance that
                // raised the exception
                // targetSecure = <SysTick instance raising exception>
            elsif HaveSysTick() == 1 then
                // SysTick target state is configurable
                targetSecure = ICSR_S.STTNS == '0';

        otherwise
            if exceptionNumber >= 16 then
                // Interrupts target the state defined by the NVIC_ITNS register
                targetSecure = NVIC_ITNS<exceptionNumber - 16> == '0';

    return targetSecure;
```

### E2.1.108 ExcInfo

```
// Exception information

type ExcInfo is (
    integer fault,              // The ID of the resulting fault, or NoFault (ie 0)
                                // if no fault occurred
    integer origFault,          // The ID if the original fault raised before
                                // escalation is considered.
    boolean isSecure,           // TRUE if the fault targets the Secure state.
    boolean origFaultIsSecure,  // TRUE if the original fault raised targeted
```

```
                                // Secure state
    boolean isTerminal,         // Set to TRUE for derived faults (eg exception on
                                // exception entry) that prevent the original
                                // exception being entered (eg a BusFault whilst
                                // fetching the exception vector address).
    boolean inExcTaken,         // TRUE if the exception occurred during ExceptionTaken()
                                // This is used to determine if the LR update and the
                                // callee stacking operations have been performed, and
                                // therefore whether the derived exception should be
                                // treated as a tail chain.
    boolean lockup,             // Set to TRUE if the exception should cause a lockup.
    boolean termInst            // Set to TRUE if the exception should cause the
                                // instruction to be terminated.
)
```

## E2.1.109 ExclusiveMonitorsPass

```
// 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';
        excInfo = CreateException(UsageFault);
    else
        (excInfo, memaddrdesc) = ValidateAddress(address, AccType_NORMAL,
                                                 FindPriv(), IsSecure(), TRUE, TRUE);
    HandleException(excInfo);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if memaddrdesc.mematttrs.shareable then
        passed = passed && IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    if passed then
        ClearExclusiveLocal(ProcessorID());
    return passed;
```

## E2.1.110 ExecBeats

```
// ExecBeats()
// ===========

boolean ExecBeats()
    // PEs are not constrained to following the beat execution pattern shown in
    // this function. Any pattern is permitted providing it meets the following
    // requirements:
    //     1) The new pattern of completed beats in representable as a valid ECI
    //        value.
    //     2) The beat execution rules are not violated (see specification).
    //     3) All ECI encodings are accepted an inputs, even if the PE can't
    //        generate that ECI value.
    newBeatComplete = BeatComplete;
    for instId = 0 to MAX_OVERLAPPING_INSTRS-1
        if _InstInfo[instId].Valid then
            _InstID                = instId;
            _CurrentInstrExecState = GetInstrExecState(instId);
            InstStateCheck(ThisInstr());
            // Find the first ticks worth of beats that isn't complete
            beatBits   = Elem[newBeatComplete, instId, MAX_BEATS];
            baseBeatId = 0;
            while Elem[beatBits, baseBeatId, BEATS_PER_TICK] == Ones(BEATS_PER_TICK) do
                baseBeatId = baseBeatId + BEATS_PER_TICK;
```

```
24
25              // Perform all the beats in this tick for the current instruction
26              for beatInTick = 0 to BEATS_PER_TICK-1
27                  beatId = baseBeatId + beatInTick;
28                  // Only perform the beat if it hasn't already been completed
29                  beatFlagIdx = (instId * MAX_BEATS) + beatId;
30                  if newBeatComplete<beatFlagIdx> == '0' then
31                      _BeatID          = beatId;
32                      _AdvanceVPTState = TRUE;
33                      cond             = DefaultCond();
34                      DecodeExecute(ThisInstr(), ThisInstrAddr(), ThisInstrLength() == 2, cond)
                            ;
35                      newBeatComplete<beatFlagIdx> = '1';
36                      // Advance the VPT state for the current beat if the instruction
37                      // didn't update the mask directly.
38                      if _AdvanceVPTState then
39                          VPTAdvance(beatId);
40
41      // If the older instruction is now complete advance the state and beat
42      // complete flags
43      commitState = newBeatComplete<MAX_BEATS-1:0> == Ones(MAX_BEATS);
44      if commitState then
45          newBeatComplete = LSR(newBeatComplete, MAX_BEATS);
46
47      // Update the beat complete flags. This is done after all the beats in the
48      // tick have been executed, as such it is not advanced if an exception
49      // terminates execution of the current tick
50      BeatComplete = newBeatComplete;
51
52      return commitState;
```

### E2.1.111 ExecuteCPCheck

```
1  // ExecuteCPCheck()
2  // ================
3
4  ExecuteCPCheck(integer cp)
5      // Check access to coprocessor is enabled
6      excInfo = CheckCPEnabled(cp);
7      HandleException(excInfo);
```

### E2.1.112 ExecuteFPCheck

```
1  // ExecuteFPCheck()
2  // ================
3
4  ExecuteFPCheck()
5      // Preserve any lazy FP state
6      PreserveFPState();
7
8      // Update the ownership of the FP context
9      FPCCR_S.S = if IsSecure() then '1' else '0';
10
11     // Update CONTROL.FPCA, and create new FP context
12     // if this has been enabled by setting FPCCR.ASPEN to 1
13     if FPCCR.ASPEN == '1' &&
14        (CONTROL.FPCA == '0' || (IsSecure() && CONTROL_S.SFPA == '0')) then
15         CONTROL.FPCA = '1';
16         if IsSecure() then
17             CONTROL_S.SFPA = '1';
18         FPSCR = FPDSCR<31:0>;
19         VPR   = Zeros();
20     return;
```

### E2.1.113 ExecutionPriority

```
// ExecutionPriority()
// ===================
// Determine the current execution priority

integer ExecutionPriority()

    boostedpri = HighestPri();       // Priority influence of BASEPRI, PRIMASK and FAULTMASK

    // Calculate boosted priority effect due to BASEPRI for both Security states
    PriSNsPri = RestrictedNSPri();
    if HaveMainExt() then
        if UInt(BASEPRI_NS<7:0>) != 0 then
            basepri = UInt(BASEPRI_NS<7:0>);
            // Include the PRIGROUP effect
            subgroupshift = UInt(AIRCR_NS.PRIGROUP);
            groupvalue    = 2 << subgroupshift;
            subgroupvalue = basepri MOD groupvalue;
            boostedpri    = basepri - subgroupvalue;
            if AIRCR_S.PRIS == '1' then
                boostedpri = (boostedpri >> 1) + PriSNsPri;

        if UInt(BASEPRI_S<7:0>) != 0 then
            basepri = UInt(BASEPRI_S<7:0>);
            // Include the PRIGROUP effect
            subgroupshift = UInt(AIRCR_S.PRIGROUP);
            groupvalue    = 2 << subgroupshift;
            subgroupvalue = basepri MOD groupvalue;
            basepri       = basepri - subgroupvalue;
            if boostedpri > basepri then
                boostedpri = basepri;

    // Calculate boosted priority effect due to PRIMASK for both Security states
    if PRIMASK_NS.PM == '1' then
        if AIRCR_S.PRIS == '0' then
            boostedpri = 0;
        else
            if boostedpri > PriSNsPri then
                boostedpri = PriSNsPri;

    if PRIMASK_S.PM == '1' then
        boostedpri = 0;

    // Calculate boosted priority effect due to FAULTMASK for both Security states
    if HaveMainExt() then
        if FAULTMASK_NS.FM == '1' then
            if AIRCR.BFHFNMINS == '0' then
                if AIRCR_S.PRIS == '0' then
                    boostedpri = 0;
                else
                    if boostedpri > PriSNsPri then
                        boostedpri = PriSNsPri;
            else
                boostedpri = -1;

        if FAULTMASK_S.FM == '1' then
            boostedpri = if AIRCR.BFHFNMINS == '0' then -1 else -3;

    // Finally calculate the resultant priority after boosting
    rawExecPri = RawExecutionPriority();
    if boostedpri < rawExecPri then
        priority = boostedpri;
    else
        priority = rawExecPri;

    assert priority IN {-4 .. 256};
    return priority;
```

### E2.1.114 Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

### E2.1.115 ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// Return TRUE if Halting debug is enabled by the IMPLEMENTATION DEFINED authentication
    interface.

boolean ExternalInvasiveDebugEnabled()
    // In the recommended interface, ExternalInvasiveDebugEnabled returns the state of
    // the DBGEN signal.
    return DBGEN == HIGH;
```

### E2.1.116 ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// Return TRUE if non-invasive debug is enabled by the IMPLEMENTATION DEFINED authentication
// interface.

boolean ExternalNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of
    // the (DBGEN OR NIDEN) signal.
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

### E2.1.117 ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// Return TRUE if Secure Halting debug is enabled by the IMPLEMENTATION DEFINED
    authentication
// interface.

boolean ExternalSecureInvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureInvasiveDebugEnabled returns the state
    // of the (DBGEN AND SPIDEN) signal.
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

### E2.1.118 ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// Return TRUE if Secure non-invasive debug is enabled by the IMPLEMENTATION DEFINED
    authentication
// interface.

boolean ExternalSecureNoninvasiveDebugEnabled()
    // In the recommended interface, ExternalSecureNoninvasiveDebugEnabled returns the
    // state of the (DBGEN OR NIDEN) AND (SPIDEN OR SPNIDEN) signal.
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

### E2.1.119 ExternalSecureSelfHostedDebugEnabled

```
// ExternalSecureSelfHostedDebugEnabled()
// ======================================
// Return TRUE if Secure self-hosted debug is enabled by the IMPLEMENTATION DEFINED
    authentication
// interface.

boolean ExternalSecureSelfHostedDebugEnabled()
    // In the recommended interface, ExternalSecureSelfHostedDebugEnabled returns the state
    // of the (DBGEN AND SPIDEN) signal.
    return DBGEN == HIGH && SPIDEN == HIGH;
```

## E2.1.120 ExtType

```
// Types of ISA extension

enumeration ExtType {ExtType_Mve,
                     ExtType_MveFp,
                     ExtType_MveOrFp,
                     ExtType_MveOrDpFp,
                     ExtType_Unknown,
                     ExtType_HpFp,
                     ExtType_SpFp,
                     ExtType_DpFp};
```

## E2.1.121 FaultNumbers

```
// Fault Numbers
// =============

// The fault numbers are a subset of ExceptionNumber and can be one of the
// following values:
constant integer NoFault      = 0;
constant integer Reset        = 1;
constant integer NMI          = 2;
constant integer HardFault    = 3;
constant integer MemManage    = 4;
constant integer BusFault     = 5;
constant integer UsageFault   = 6;
constant integer SecureFault  = 7;
constant integer SVCall       = 11;
constant integer DebugMonitor = 12;
constant integer PendSV       = 14;
constant integer SysTick      = 15;
```

## E2.1.122 FetchInstr

```
// FetchInstr()
// ============

(bits(32), boolean) FetchInstr(bits(32) addr)
    // NOTE: It is CONSTRAINED UNPREDICTABLE whether otherwise valid sequential
    //       instruction fetches that cross from Non-secure to Secure memory
    //       generate a INVEP SecureFault, or transition normally.
    sgOpcode = 0xE97FE97F<31:0>;

    hw1Attr  = SecurityCheck(addr, TRUE, IsSecure());
    // Fetch the a T16 instruction, or the first half of a T32.
    hw1Instr = MemI[addr];

    // If the T bit is clear then the instruction can't be decoded
    if EPSR.T == '0' then
        // Attempted NS->S domain crossings with the T bit clear raise an INVEP
        // SecureFault
        if !IsSecure() && !hw1Attr.ns then
            SFSR.INVEP = '1';
```

```
            excInfo = CreateException(SecureFault);
        else
            UFSR.INVSTATE = '1';
            excInfo = CreateException(UsageFault);
        HandleException(excInfo);

    // Implementations are permitted to terminate the fetch process early if a
    // domain crossing is being attempted and the first 16bits of the opcode
    // isn't the first part of the SG instruction.
    if boolean IMPLEMENTATION_DEFINED "Early SG check" then
        if !IsSecure() && !hw1Attr.ns && (hw1Instr != sgOpcode<31:16>) then
            SFSR.INVEP = '1';
            excInfo = CreateException(SecureFault);
            HandleException(excInfo);

    // NOTE: Implementations are also permitted to terminate the fetch process
    //       at this point with an UNDEFINSTR UsageFault if the first 16bit is
    //       an undefined T32 prefix.

    // If the data fetched is the top half of a T32 instruction fetch the bottom
    // 16 bits
    isT16 = UInt(hw1Instr<15:11>) < UInt('11101');
    if isT16 then
        instr = Zeros(16) : hw1Instr;
    else
        hw2Attr = SecurityCheck(addr+2, TRUE, IsSecure());
        // The following test covers 2 possible fault conditions:-
        //  1) NS code branching to a T32 instruction where the first half is in
        //     NS memory, and the second half is in S memory.
        //  2) NS code branching to a T32 instruction in S & NSC memory, but
        //     where the second half of the instruction is in NS memory.
        if !IsSecure() && (hw1Attr.ns != hw2Attr.ns) then
            SFSR.INVEP = '1';
            excInfo = CreateException(SecureFault);
            HandleException(excInfo);

        // Fetch the second half of T32 instruction
        instr   = hw1Instr : MemI[addr+2];

    // Raise a fault if an otherwise valid NS->S transition that doesn't land on
    // an SG instruction.
    if !IsSecure() && !hw1Attr.ns && (instr != sgOpcode) then
        SFSR.INVEP = '1';
        excInfo = CreateException(SecureFault);
        HandleException(excInfo);
    return (instr, isT16);
```

### E2.1.123 FindPriv

```
// FindPriv()
// ==========

boolean FindPriv()
    return CurrentModeIsPrivileged();
```

### E2.1.124 FixedToFP

```
// FixedToFP()
// ===========

bits(N) FixedToFP(bits(M) operand, integer N, integer fraction_bits, boolean unsigned,
                  boolean round_to_nearest, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_to_nearest then fpscr_val.RMode = FPSCR_RMode_RN;
    int_operand = if unsigned then UInt(operand) else SInt(operand);
    real_operand = Real(int_operand) / 2.0^fraction_bits;
```

```
    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, N, fpscr_val);
    return result;
```

### E2.1.125 FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) operand)
    assert N IN {16,32,64};
    return '0' : operand<N-2:0>;
```

### E2.1.126 FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding
                mode
                result_sign = if fpscr_val.RMode == FPSCR_RMode_RM then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);
    return result;
```

### E2.1.127 FPB_CheckBreakPoint

```
// FPB_CheckBreakPoint
// ===================
// Check for Flash Patch Break point

boolean FPB_CheckBreakPoint(bits(32) iaddr, integer size, boolean is_ifetch, boolean
    is_secure)

    match = FPB_CheckMatchAddress(iaddr);
    if !match && size == 4 && FPB_CheckMatchAddress(iaddr + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_FPBreakpoint);
    return match;
```

### E2.1.128 FPB_CheckMatchAddress

```
// FPB_CheckMatchAddress
// =====================
// Flash Patch breakpoint instruction address comparison

boolean FPB_CheckMatchAddress(bits(32) iaddr)

    if FP_CTRL.ENABLE == '0' then return FALSE; // FPB not enabled

    // Instruction Comparator.
    num_addr_cmp = UInt(FP_CTRL.NUM_CODE);
    if num_addr_cmp == 0 then return FALSE;     // No comparator support

    for N = 0 to (num_addr_cmp - 1)
        if FP_COMP[N].BE == '1' then            // Breakpoint enabled
            if iaddr<31:1> == FP_COMP[N].BPADDR then
                return TRUE;

    return FALSE;
```

## E2.1.129 FPCompare

```
// FPCompare()
// ===========

(bit, bit, bit, bit) FPCompare(bits(N) op1, bits(N) op2, boolean quiet_nan_exc,
                               boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,-,value1) = FPUnpack(op1, fpscr_val);
    (type2,-,value2) = FPUnpack(op2, fpscr_val);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN
        then
        result = ('0','0','1','1');
        if type1==FPType_SNaN || type2==FPType_SNaN || quiet_nan_exc then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = ('0','1','1','0');
        elsif value1 < value2 then
            result = ('1','0','0','0');
        else  // value1 > value2
            result = ('0','0','1','0');
    return result;
```

## E2.1.130 FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    assert N IN {16,32,64};
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

## E2.1.131 FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
```

```
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (fp_type1 == FPType_Infinity);  inf2 = (fp_type2 == FPType_Infinity);
        zero1 = (fp_type1 == FPType_Zero);     zero2 = (fp_type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpscr_val);
        elsif zero1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1/value2, N, fpscr_val);
    return result;
```

### E2.1.132 FPDoubleToHalf

```
// FPDoubleToHalf()
// ================
bits(16) FPDoubleToHalf(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val.AHP == '1' then
            result = FPZero(sign, 16);
        elsif fpscr_val.DN == '1' then
            result = FPDefaultNaN(16);
        else
            result = sign : '11111 1' : operand<50:42>;
        if fp_type == FPType_SNaN || fpscr_val.AHP == '1' then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        if fpscr_val.AHP == '1' then
            result = sign : Ones(15);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        else
            result = FPInfinity(sign, 16);
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, 16);
    else
        result = FPRoundCV(value, 16, fpscr_val);
    return result;
```

### E2.1.133 FPDoubleToSingle

```
// FPDoubleToSingle()
// ==================

bits(32) FPDoubleToSingle(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val.DN == '1' then
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<50:29>;
        if fp_type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        result = FPInfinity(sign, 32);
```

```
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRoundCV(value, 32, fpscr_val);
    return result;
```

### E2.1.134 FPExc

```
// Floating point exceptions

enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                   FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

### E2.1.135 FPHalfToDouble

```
// FPHalfToDouble()
// ================

bits(64) FPHalfToDouble(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val.DN == '1' then
            result = FPDefaultNaN(64);
        else
            result = sign : '11111111111 1' : operand<8:0> : Zeros(42);
        if fp_type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        result = FPInfinity(sign, 64);
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, 64);
    else
        result = FPRoundCV(value, 64, fpscr_val); // Rounding will be exact
    return result;
```

### E2.1.136 FPHalfToSingle

```
// FPHalfToSingle()
// ================

bits(32) FPHalfToSingle(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val.DN == '1' then
            result = FPDefaultNaN(32);
        else
            result = sign : '11111111 1' : operand<8:0> : Zeros(13);
        if fp_type == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        result = FPInfinity(sign, 32);
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRoundCV(value, 32, fpscr_val); // Rounding will be exact
    return result;
```

### E2.1.137 FPInfinity

```
// FPInfinity()
// ============

```

```
bits(N) FPInfinity(bit sign, integer N)
    assert N IN {16,32,64};
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

### E2.1.138 FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
    if !done then
        if value1 > value2 then
            (fp_type,sign,value) = (fp_type1,sign1,value1);
        else
            (fp_type,sign,value) = (fp_type2,sign2,value2);
        if fp_type == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fp_type == FPType_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign, N);
        else
            result = FPRound(value, N, fpscr_val);
    return result;
```

### E2.1.139 FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N IN {16,32,64};
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

### E2.1.140 FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};

    (type1,-,-) = FPUnpack(op1, FPSCR);
    (type2,-,-) = FPUnpack(op2, FPSCR);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1', N);
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1', N);

    return FPMax(op1, op2, TRUE);
```

### E2.1.141 FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
    if !done then
        if value1 < value2 then
            (fp_type,sign,value) = (fp_type1,sign1,value1);
        else
            (fp_type,sign,value) = (fp_type2,sign2,value2);
        if fp_type == FPType_Infinity then
            result = FPInfinity(sign, N);
        elsif fp_type == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign, N);
        else
            result = FPRound(value, N, fpscr_val);
    return result;
```

### E2.1.142 FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};

    (fp_type1,-,-) = FPUnpack(op1, FPSCR);
    (fp_type2,-,-) = FPUnpack(op2, FPSCR);

    // Treat a single quiet-NaN as +Infinity
    if fp_type1 == FPType_QNaN && fp_type2 != FPType_QNaN then
        op1 = FPInfinity('0', N);
    elsif fp_type1 != FPType_QNaN && fp_type2 == FPType_QNaN then
        op2 = FPInfinity('0', N);

    return FPMin(op1, op2, TRUE);
```

### E2.1.143 FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif inf1 || inf2 then
            result_sign = if sign1 == sign2 then '0' else '1';
            result = FPInfinity(result_sign, N);
        elsif zero1 || zero2 then
            result_sign = if sign1 == sign2 then '0' else '1';
```

```
            result = FPZero(result_sign, N);
        else
            result = FPRound(value1*value2, N, fpscr_val);
    return result;
```

### E2.1.144 FPMulAdd

```
// FPMulAdd()
// ==========
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
                 boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (typeA,signA,valueA) = FPUnpack(addend, fpscr_val);
    (type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpscr_val);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN(N);
        FPProcessException(FPExc_InvalidOp, fpscr_val);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = if sign1 == sign2 then '0' else '1';
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA == NOT(signP)) then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding
                mode
                result_sign = if fpscr_val.RMode == FPSCR_RMode_RM then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);

    return result;
```

### E2.1.145 FPNeg

```
// FPNeg()
```

```
// =======

bits(N) FPNeg(bits(N) operand)
    assert N IN {16,32,64};
    return NOT(operand<N-1>) : operand<N-2:0>;
```

### E2.1.146 FPProcessException

```
// FPProcessException()
// ====================
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

FPProcessException(FPExc exception, FPSCR_Type fpscr_val)
    // Get appropriate FPSCR bit numbers
    case exception of
        when FPExc_InvalidOp     enable = 8;   cumul = 0;
        when FPExc_DivideByZero  enable = 9;   cumul = 1;
        when FPExc_Overflow      enable = 10;  cumul = 2;
        when FPExc_Underflow     enable = 11;  cumul = 3;
        when FPExc_Inexact       enable = 12;  cumul = 4;
        when FPExc_InputDenorm   enable = 15;  cumul = 7;
    if fpscr_val<enable> == '1' then
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    else
        FPSCR<cumul> = '1';
    return;
```

### E2.1.147 FPProcessNaN

```
// FPProcessNaN()
// ==============
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

bits(N) FPProcessNaN(FPType fp_type, bits(N) operand, FPSCR_Type fpscr_val)
    assert N IN {16,32,64};
    if N == 16 then    topfrac = 9;
    elsif N == 32 then topfrac = 22;
    else               topfrac = 51;
    result = operand;
    if fp_type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    if fpscr_val.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(N);
    return result;
```

### E2.1.148 FPProcessNaNs

```
// FPProcessNaNs()
// ===============
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 bits(32) fpscr_val)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
```

```
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

### E2.1.149 FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  bits(32) fpscr_val)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpscr_val);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpscr_val);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpscr_val);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpscr_val);
    else
        done = FALSE;  result = Zeros(N);  // 'Don't care' result
    return (done, result);
```

### E2.1.150 FPRound

```
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real value, integer N, FPSCR_Type fpscr_val)
    fpscr_val.AHP = '0';
    return FPRoundBase(value, N, fpscr_val);
```

### E2.1.151 FPRoundBase

```
// FPRoundBase()
// =============
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

bits(N) FPRoundBase(real value, integer N, FPSCR_Type fpscr_val)
    assert N IN {16,32,64};
    assert value != 0.0;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
```

```
    minimum_exp = 2 - 2^(E-1);
    constant integer F = N - E - 1;

    // Split value into sign, unrounded mantissa and exponent.
    if value < 0.0 then
        sign = '1';  mantissa = -value;
    else
        sign = '0';  mantissa = value;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((N != 16 && fpscr_val.FZ == '1') || (N == 16 && fpscr_val.FZ16 == '1')) && exponent <
         minimum_exp then
        result = FPZero(sign, N);
        FPSCR.UFC = '1';  // Flush-to-zero never generates a trapped exception
    else

        // Start creating the exponent value for the result. Start by biasing the actual
            exponent
        // so that the minimum exponent becomes 1, lower values 0 (indicating possible
            underflow).
        biased_exp = Max(exponent - minimum_exp + 1, 0);
        if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

        // Get the unrounded mantissa as an integer, and the "units in last place" rounding
            error.
        int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if
            not
        error = mantissa * 2.0^F - Real(int_mant);

        // Underflow occurs if exponent is too small before rounding, and result is inexact
            or
        // the Underflow exception is trapped.
        if biased_exp == 0 && error != 0.0 then
            FPProcessException(FPExc_Underflow, fpscr_val);

        // Round result according to rounding mode.
        case fpscr_val.RMode of
            when FPSCR_RMode_RN  // Round to Nearest (rounding to even if exactly halfway)
                round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
                overflow_to_inf = TRUE;
            when FPSCR_RMode_RP  // Round towards Plus Infinity
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when FPSCR_RMode_RM  // Round towards Minus Infinity
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when FPSCR_RMode_RZ  // Round towards Zero
                round_up = FALSE;
                overflow_to_inf = FALSE;
        if round_up then
            int_mant = int_mant + 1;
            if int_mant == 2^F then      // Rounded up from denormalized to normalized
                biased_exp = 1;
            if int_mant == 2^(F+1) then  // Rounded up to next exponent
                biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

        // Deal with overflow and generate result.
        if N != 16 || fpscr_val.AHP == '0' then // Single, double or IEEE half precision
            if biased_exp >= 2^E - 1 then
                result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N
                    );
                FPProcessException(FPExc_Overflow, fpscr_val);
                error = 1.0;  // Ensure that an Inexact exception occurs
            else
```

```
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;
        else                                    // Alternative half precision
            if biased_exp >= 2^E then
                result = sign : Ones(N-1);
                FPProcessException(FPExc_InvalidOp, fpscr_val);
                error = 0.0; // Ensure that an Inexact exception does not occur
            else
                result = sign : biased_exp<E-1:0> : int_mant<F-1:0>;

        // Deal with Inexact exception.
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpscr_val);

    return result;
```

### E2.1.152 FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data processing operations the FZ16 bit
// is ignored and the AHP bit is observed.

bits(N) FPRoundCV(real value, integer N, FPSCR_Type fpscr_val)
    fpscr_val.FZ16 = '0';
    return FPRoundBase(value, N, fpscr_val);
```

### E2.1.153 FPRoundInt

```
// FPRoundInt()
// ============
// Round floating-point value to nearest integral floating point value
// using given rounding mode. If exact is TRUE, set inexact flag if result
// is not numerically equal to given value.

bits(N) FPRoundInt(bits(N) op, bits(2) rmode, boolean away, boolean exact)
    assert N IN {16,32,64};

    // Unpack using FPSCR to determine if subnormals are flushed-to-zero
    (fp_type,sign,value) = FPUnpack(op, FPSCR);

    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        result = FPProcessNaN(fp_type, op, FPSCR);
    elsif fp_type == FPType_Infinity then
        result = FPInfinity(sign, N);
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, N);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rmode of
            when '00'  // Round to nearest, ties to even
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when '01'  // Round towards Plus Infinity
                round_up = (error != 0.0);
            when '10'  // Round towards Minus Infinity
                round_up = FALSE;
            when '11'  // Round towards Zero
                round_up = (error != 0.0 && int_result < 0);

        if away then  // Round towards Zero, ties away
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;
```

```
39
40          // Convert integer value into an equivalent real value
41          real_result = Real(int_result);
42
43          // Re-encode as a floating-point value, result is always exact
44          if real_result == 0.0 then
45              result = FPZero(sign, N);
46          else
47              result = FPRound(real_result, N, FPSCR);
48
49          // Generate inexact exceptions
50          if error != 0.0 && exact then
51              FPProcessException(FPExc_Inexact, FPSCR);
52
53      return result;
```

### E2.1.154 FPSingleToDouble

```
1   // FPSingleToDouble()
2   // ==================
3
4   bits(64) FPSingleToDouble(bits(32) operand, boolean fpscr_controlled)
5       fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6       (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
7       if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8           if fpscr_val.DN == '1' then
9               result = FPDefaultNaN(64);
10          else
11              result = sign : '11111111111 1' : operand<21:0> : Zeros(29);
12          if fp_type == FPType_SNaN then
13              FPProcessException(FPExc_InvalidOp, fpscr_val);
14      elsif fp_type == FPType_Infinity then
15          result = FPInfinity(sign, 64);
16      elsif fp_type == FPType_Zero then
17          result = FPZero(sign, 64);
18      else
19          result = FPRoundCV(value, 64, fpscr_val);  // Rounding will be exact
20      return result;
```

### E2.1.155 FPSingleToHalf

```
1   // FPSingleToHalf()
2   // ================
3
4   bits(16) FPSingleToHalf(bits(32) operand, boolean fpscr_controlled)
5       fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
6       (fp_type,sign,value) = FPUnpackCV(operand, fpscr_val);
7       if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
8           if fpscr_val.AHP == '1' then
9               result = FPZero(sign, 16);
10          elsif fpscr_val.DN == '1' then
11              result = FPDefaultNaN(16);
12          else
13              result = sign : '11111 1' : operand<21:13>;
14          if fp_type == FPType_SNaN || fpscr_val.AHP == '1' then
15              FPProcessException(FPExc_InvalidOp, fpscr_val);
16      elsif fp_type == FPType_Infinity then
17          if fpscr_val.AHP == '1' then
18              result = sign : Ones(15);
19              FPProcessException(FPExc_InvalidOp, fpscr_val);
20          else
21              result = FPInfinity(sign, 16);
22      elsif fp_type == FPType_Zero then
23          result = FPZero(sign, 16);
24      else
25          result = FPRoundCV(value, 16, fpscr_val);
26      return result;
```

### E2.1.156 FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) operand)
    assert N IN {16,32,64};
    (fp_type,sign,value) = FPUnpack(operand, FPSCR);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        result = FPProcessNaN(fp_type, operand, FPSCR);
    elsif fp_type == FPType_Zero then
        result = FPZero(sign, N);
    elsif fp_type == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(N);
        FPProcessException(FPExc_InvalidOp, FPSCR);
    else
        result = FPRound(Sqrt(value), N, FPSCR);
    return result;
```

### E2.1.157 FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type1,sign1,value1) = FPUnpack(op1, fpscr_val);
    (fp_type2,sign2,value2) = FPUnpack(op2, fpscr_val);
    (done,result) = FPProcessNaNs(fp_type1, fp_type2, op1, op2, fpscr_val);
    if !done then
        inf1 = (fp_type1 == FPType_Infinity);  inf2 = (fp_type2 == FPType_Infinity);
        zero1 = (fp_type1 == FPType_Zero);     zero2 = (fp_type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(N);
            FPProcessException(FPExc_InvalidOp, fpscr_val);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding
                mode
                result_sign = if fpscr_val.RMode == FPSCR_RMode_RM then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);
    return result;
```

### E2.1.158 FPToFixed

```
// FPToFixed()
// ===========

bits(M) FPToFixed(bits(N) operand, integer M, integer fraction_bits, boolean unsigned,
                  boolean round_towards_zero, boolean fpscr_controlled)
    assert N IN {16,32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_towards_zero then fpscr_val.RMode = FPSCR_RMode_RZ;
    (fp_type,-,value) = FPUnpack(operand, fpscr_val);

```

```
    // For NaNs and infinities, FPUnpack() has produced a value that will round to the
    // required result of the conversion. Also, the value produced for infinities will
    // cause the conversion to overflow and signal an Invalid Operation floating-point
    // exception as required. NaNs must also generate such a floating-point exception.
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpscr_val);

    // Scale value by specified number of fraction bits, then start rounding to an integer
    // and determine the rounding error.
    value = value * 2.0^fraction_bits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Apply the specified rounding mode.
    case fpscr_val.RMode of
        when FPSCR_RMode_RN  // Round to Nearest (rounding to even if exactly halfway)
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPSCR_RMode_RP  // Round towards Plus Infinity
            round_up = (error != 0.0);
        when FPSCR_RMode_RM  // Round towards Minus Infinity
            round_up = FALSE;
        when FPSCR_RMode_RZ  // Round towards Zero
            round_up = (error != 0.0 && int_result < 0);
    if round_up then int_result = int_result + 1;

    // Bitstring result is the integer result saturated to the destination size, with
    // saturation indicating overflow of the conversion (signaled as an Invalid
    // Operation floating-point exception).
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpscr_val);

    return result;
```

### E2.1.159 FPToFixedDirected

```
// FPToFixedDirected()
// ===================

bits(M) FPToFixedDirected(bits(N) op, integer fbits, boolean unsigned,
                          bits(2) round_mode, boolean fpscr_controlled)
    assert N IN {16,32,64};

    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();

    // Unpack using FPSCR to determine if subnormals are flushed-to-zero
    (fp_type,-,value) = FPUnpack(op, fpscr_val);

    // If NaN, set cumulative flag or take exception
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, FPSCR);

    // Scale by fractional bits and produce integer rounded towards
    // minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case round_mode of
        when '00' // ties away
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));
        when '01' // nearest even
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when '10' // plus infinity
            round_up = (error != 0.0);
```

```
        when '11' // neg infinity
          round_up = FALSE;

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);

    if overflow then
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpscr_val);
    return result;
```

### E2.1.160 FPType

```
// Type of floating-point value. Floating-point values are categorized into one
// of the following type during unpacking.

enumeration FPType {FPType_Nonzero, FPType_Zero, FPType_Infinity, FPType_QNaN, FPType_SNaN};
```

### E2.1.161 FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPSCR_Type fpscr_val)
    fpscr_val.AHP = '0';
    return FPUnpackBase(fpval, fpscr_val);
```

### E2.1.162 FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPSCR_Type fpscr_val)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpscr_val.FZ16 == '1' then
                fp_type = FPType_Zero;  value = 0.0;
            else
                fp_type = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpscr_val.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fp_type = FPType_Infinity;  value = 2.0^1000000;
            else
                fp_type = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
```

```
                value = 0.0;
        else
            fp_type = FPType_Nonzero;
            value   = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpscr_val.FZ == '1' then
                fp_type = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                fp_type = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fp_type = FPType_Infinity;  value = 2.0^1000000;
            else
                fp_type = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fp_type = FPType_Nonzero;
            value   = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpscr_val.FZ == '1' then
                fp_type = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpscr_val);
            else
                fp_type = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52)
                    ;
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fp_type = FPType_Infinity;  value = 2.0^1000000;
            else
                fp_type = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fp_type = FPType_Nonzero;
            value   = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (fp_type, sign, value);
```

### E2.1.163 FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPSCR_Type fpscr_val)
    fpscr_val.FZ16 = '0';
    return FPUnpackBase(fpval, fpscr_val);
```

### E2.1.164 FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N IN {16,32,64};
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

### E2.1.165 FunctionReturn

```
// FunctionReturn()
// ================

ExcInfo FunctionReturn()
    exc = DefaultExcInfo();

    // Pull the return address and IPSR off the Secure stack
    mode     = CurrentMode();
    spName   = LookUpSP_with_security_mode(TRUE, mode);
    framePtr = _SP(spName);
    if !IsAligned(framePtr, 8) then UNPREDICTABLE;
    // Only stack locations, not the load order are architected
    RETPSR_Type newPSR;
    if exc.fault == NoFault then (exc, newPSR) = Stack(framePtr, 4, spName, mode);
    if exc.fault == NoFault then (exc, newPC)  = Stack(framePtr, 0, spName, mode);

    // Check the IPSR value that has been unstacked is consistent with the current
    // mode, and being originally called from the Secure state.
    // NOTE: It is IMPLEMENTATION DEFINED whether this check is performed before
    //       or after the load of the return address above.
    if (exc.fault == NoFault) &&
       !(((IPSR.Exception == 0<8:0>) && (newPSR.Exception == 0<8:0>)) ||
         ((IPSR.Exception == 1<8:0>) && (newPSR.Exception != 0<8:0>))) then
        if HaveMainExt() then
            UFSR_S.INVPC = '1';
        // Create the exception. NOTE: If Main Extension not implemented then the fault
        // always escalates to a HardFault
        exc = CreateException(UsageFault, TRUE, TRUE);
    // The IPSR value is set as UNKNOWN if the IPSR value is not supported by the PE
    excNum = UInt(newPSR.Exception);
    validIPSR = excNum IN {0, 1, NMI, HardFault, SVCall, PendSV, SysTick};
    if !validIPSR && HaveMainExt() then
        validIPSR = excNum IN {MemManage, BusFault, UsageFault, SecureFault, DebugMonitor};
    if !validIPSR && !IsIrqValid(excNum) then
        newPSR.Exception = bits(9) UNKNOWN;

    // Only consume the function return stack frame and update the XPSR/PC if no
    // faults occured.
    if exc.fault == NoFault then
        // Transition to the Secure state
        CurrentState = SecurityState_Secure;
        // Update stack pointer. NOTE: Stack pointer limit not checked on function
        // return as stack pointer guaranteed to be ascending not descending.
        exc = _SP(spName, FALSE, TRUE, framePtr + 8);
        assert exc.fault == NoFault;

        IPSR.Exception = newPSR.Exception;
        CONTROL_S.SFPA = newPSR.SFPA;
        // IT/ICI/ECI/LOB data cleared to prevent Non-secure code interfering with
        // Secure execution
        if HaveMainExt() then
            ITSTATE = Zeros(8);
```

```
        if HaveLOBExt() then
            LO_BRANCH_INFO.VALID = '0';
        // if EPSR.T == 0, a UsageFault('Invalid State') or a HardFault is taken
        // on the next instruction depending on whether the Main Extension is
        // is implemented or not.
        EPSR.T = newPC<0>;
        BranchTo(newPC<31:1>:'0');
    return exc;
```

### E2.1.166 GenerateCoprocessorException

```
// GenerateCoprocessorException()
// ==============================

GenerateCoprocessorException()
    UFSR.UNDEFINSTR = '1';
    excInfo = CreateException(UsageFault);
    HandleException(excInfo);
```

### E2.1.167 GenerateDebugEventResponse

```
// GenerateDebugEventResponse()
// ============================
// Generate a debug event response based on the PE configuration.

GenerateDebugEventResponse(boolean isBKPT)
    if CanHaltOnEvent(IsSecure()) then
        if isBKPT then
            DFSR.BKPT = '1';
        else
            DFSR.EXTERNAL = '1';
        DHCSR.C_HALT = '1';

    elsif isBKPT then
        excInfo = CreateException(DebugMonitor);
        if excInfo.fault == DebugMonitor then
            DFSR.BKPT = '1';
        HandleException(excInfo);

    elsif CanPendMonitorOnEvent(IsSecure(), TRUE, TRUE) then
        DFSR.EXTERNAL  = '1';
        DEMCR.MON_PEND = '1';
```

### E2.1.168 GenerateIntegerZeroDivide

```
// GenerateIntegerZeroDivide()
// ===========================

GenerateIntegerZeroDivide()
    UFSR.DIVBYZERO = '1';
    excInfo = CreateException(UsageFault);
    HandleException(excInfo);
```

### E2.1.169 GetActiveChains

```
// GetActiveChains()
// =================

integer GetActiveChains()
    count = 0;
    if HaveMve() then
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if _InstInfo[i].Valid then
                count = count + 1;
    return count;
```

### E2.1.170 GetCurInstrBeat

```
// GetCurInstrBeat()
// =================

(integer, bits(4)) GetCurInstrBeat()
    assert HaveMve();
    // By default assume all lanes are active
    elmtMask = Ones(4);

    // If VPT active apply the predicate flags in VPR.P0
    if VPTActive() then
        elmtMask = elmtMask AND Elem[VPR.P0, _BeatID, 4];

    // LOB truncation may override the flags on the last iteration of a loop
    // LTPSIZE < 4 is a proxy for knowing if we're in a loop and tail predication is active.
    ltpsize = if _CurrentInstrExecState.ResetLTPSize then 4 else LTPSIZE;
    if ltpsize < 4 && IsLastLowOverheadLoop() then
        loopCount = _CurrentInstrExecState.LoopCount;
        predSize  = ltpsize;
        fullMask  = ZeroExtend(Ones(UInt(loopCount<4-predSize:0> : Zeros(predSize))), 16);
        elmtMask  = elmtMask AND Elem[fullMask, _BeatID, 4];
    return (_BeatID, elmtMask);
```

### E2.1.171 GetInstrExecState

```
// GetInstrExecState()
// ===================

INSTR_EXEC_STATE_Type GetInstrExecState(integer next)
    // next = 0: returns current (committed) state
    // next > 0: returns n-th state from now
    assert (next >= 0 && next < MAX_BEATS);
    INSTR_EXEC_STATE_Type state;

    // 1) Next == 0: current committed state
    state.FetchAddr         = _RName[RNamesPC];
    state.ITState           = EPSR.IT;
    state.L                 = '0';
    state.T16IND            = '0';
    state.LoopCount         = LR;
    state.LOBranchInfoValid = LO_BRANCH_INFO.VALID;
    state.ResetLTPSize      = FALSE;

    // 2) Determine speculative future
    for i = 1 to next
        // Handle normal PC changes BEFORE LOB handling
        if _PCChanged && i == 1 then
            state.FetchAddr = _NextInstrAddr;
        else
            state.FetchAddr = state.FetchAddr + ThisInstrLength(i-1);

        // If the IT state has been directly modified return that value as the
        // next state, otherwise advance the IT state normally.
        if _ITStateChanged && i == 1 then
            state.ITState = _NextInstrITState;
        else
            state.ITState = ITAdvance(state.ITState);

        // Check if loop or branch triggers PC change (unless normal PC change)
        if (!_PCChanged) && HaveLOBExt() then
            state = HandleLO(state);

    return state;
```

### E2.1.172 Halt

```
// Halt()
// ======

Halt()
    // Halt
    Halted = TRUE;

    // Clear lockup state
    LockedUp = FALSE;

    // Upon entering debug state, S_REGRDY becomes valid hence must be set to '1'.
    DHCSR.S_REGRDY = '1';

    // Any pending return operation is cleared and can be re-pended on
    // exit from Debug State.
    _PendingReturnOperation = FALSE;

    // Clear all remaining in flight instructions
    for i = 0 to MAX_OVERLAPPING_INSTRS-1
        _InstInfo[i].Valid = FALSE;
```

### E2.1.173 Halted

```
// Indicates the PE is in Debug State

boolean Halted;
```

### E2.1.174 HaltingDebugAllowed

```
// HaltingDebugAllowed()
// =====================

boolean HaltingDebugAllowed()
    return ExternalInvasiveDebugEnabled() || Halted;
```

### E2.1.175 HandleException

```
// HandleException()
// =================

HandleException(ExcInfo excInfo)
    if excInfo.fault != NoFault then
        if excInfo.lockup then
            Lockup(excInfo.termInst);
        else
            // If the fault escalated to a HardFault update the syndrome info
            if HaveMainExt() && excInfo.fault == HardFault then
                if excInfo.origFault == DebugMonitor then
                    HFSR.DEBUGEVT = '1';
                elsif excInfo.origFault != HardFault then
                    HFSR.FORCED   = '1';

            // If the exception does not cause a lockup set the exception pending
            // and potentially terminate execution of the current instruction
            SetPending(excInfo.fault, excInfo.isSecure, TRUE);
            if excInfo.termInst then
                EndOfInstruction();
```

### E2.1.176 HandleExceptionTransitions

```
// HandleExceptionTransitions()
// ============================

boolean HandleExceptionTransitions(boolean commitState)
```

```
    // Check for, and process any exception returns that were requested. This
    // must be done after the instruction has completed so any exceptions
    // raised during the exception return do not interfere with the execution of
    // the instruction that cause the exception return (eg a POP causing an
    // excReturn value to be written to the PC must adjust SP even if the
    // exception return caused by the POP raises a fault).
    excRetFault               = FALSE;
    tailChainedException      = FALSE;
    EXC_RETURN_Type excReturn = NextInstrAddr();
    if _PendingReturnOperation then
        _PendingReturnOperation = FALSE;
        (excInfo, excReturn, tailChainedException) = ExceptionReturn(excReturn);
        // Handle any faults raised during exception return
        if excInfo.fault != NoFault then
            excRetFault = TRUE;
            // Either lockup, or pend the fault if it can be taken
            if excInfo.lockup then
                // Check if the fault occurred on exception return, or whether it
                // occurred during a tail chained exception entry. This is
                // because Lockups on exception return have to be handled
                // differently.
                if !excInfo.inExcTaken then
                    // If the fault occurred during exception return then the
                    // register state is UNKNOWN. This is due to the fact that
                    // an unknown amount of the exception stack frame might have
                    // been restored.
                    for n = 0 to 12
                        R[n] = bits(32) UNKNOWN;
                    LR   = bits(32) UNKNOWN;
                    XPSR = bits(32) UNKNOWN;
                    if HaveMveOrFPExt() then
                        InvalidateFPRegs(FALSE, TRUE);
                    // If lockup is entered as a result of an exception return
                    // fault the original exception is deactivated. Therefore
                    // the stack pointer must be updated to consume the
                    // exception stack frame to keep the stack depth consistent
                    // with the number of active exceptions. NOTE: The XPSR SP
                    // alignment flag is UNKNOWN, assume is was zero.
                    ConsumeExcStackFrame(excReturn, '0');
                    // IPSR from stack is UNKNOWN, set IPSR based on mode
                    // specified in EXC_RETURN.
                    IPSR.Exception = (if excReturn.Mode == '1' then NoFault else HardFault)
                        <8:0>;
                    if HaveMveOrFPExt() then
                        CONTROL.FPCA   = NOT(excReturn.FType);
                        CONTROL_S.SFPA = bit UNKNOWN;
                Lockup(FALSE);
            else
                // Set syndrome if fault escalated to a HardFault
                if HaveMainExt() &&
                   (excInfo.fault     == HardFault) &&
                   (excInfo.origFault != HardFault) then
                    HFSR.FORCED = '1';
                SetPending(excInfo.fault, excInfo.isSecure, TRUE);

    // If there is a pending exception with sufficient priority take it now.
    // This is done before committing PC and ITSTATE changes caused by the
    // previous instruction so that the committed architecture state reflects
    // the context the instruction was executed in.
    (takeException, exception, excIsSecure) = PendingExceptionDetails();
    if takeException then
        // If a fault occurred during an exception return then the exception
        // stack frame will already be on the stack, as a result entry to the
        // next exception is treated as if it were a tail chain.
        pePriority  = ExecutionPriority();
        peException = UInt(IPSR.Exception);
        peIsSecure  = IsSecure();
        if excRetFault then
            // If the fault occurred during ExceptionTaken() then LR will have
```

```
            // been updated with the new exception return value. To excReturn
            // consistent with the state of the exception stack frame we need to
            // use the updated version in this case. If no updates have occurred
            // then the excReturn value from the previous exception return is
            // used.
            if excInfo.inExcTaken then
                excReturn = LR;
            (excInfo, excReturn) = TailChain(exception, excIsSecure, excReturn);
        else
            (excInfo, excReturn) = ExceptionEntry(exception, excIsSecure, commitState);
        // Handle any derived faults that have occurred
        if excInfo.fault != NoFault then
            DerivedLateArrival(pePriority, peException, peIsSecure, excInfo,
                               exception, excIsSecure, excReturn);

    return takeException || tailChainedException;
```

## E2.1.177 HandleLO

```
// HandleLO()
// ==========

INSTR_EXEC_STATE_Type HandleLO(INSTR_EXEC_STATE_Type state)
    // The default state for the link bit is FALSE
    state.L = '0';

    // If valid branch info matches the fetch address update the LOB state and
    // fetch address accordingly.
    if state.LOBranchInfoValid == '1' then
        if LO_BRANCH_INFO.END_ADDR == state.FetchAddr<31:1> then
            state.L      = LO_BRANCH_INFO.BF AND LO_BRANCH_INFO.LF;
            state.T16IND = LO_BRANCH_INFO.T16IND;
            // Conditions for LOB handling in an IT block
            if InITBlock(state.ITState) then
                // The BF b_label is allowed to be the last instruction in an IT block.
                // As the BF branch occurs before this instruction is executed, the ITSTATE
                // needs to be updated as if the end of the IT block had been reached.
                if LO_BRANCH_INFO.BF == '1' then
                    state.ITState = Zeros(8);
                else
                    // If LO_BRANCH_INFO is valid and a low overhead branch is handled,
                    // then the behavior is CONSTRAINED UNPREDICTABLE.
                    CONSTRAINED_UNPREDICTABLE;
            // Branch cache address matched, branch to offset specified
            if  LO_BRANCH_INFO.BF == '1'                                ||
               (LO_BRANCH_INFO.BF == '0' && LO_BRANCH_INFO.LF == '1') ||
               !IsLastLowOverheadLoop(state) then
                state.FetchAddr = LO_BRANCH_INFO.JUMP_ADDR:'0';
            // If the branch is due to a BF instruction invalidate the branch
            // info so spurious branches don't occur.
            if LO_BRANCH_INFO.BF == '1' then
                state.LOBranchInfoValid = '0';
            elsif LO_BRANCH_INFO.LF == '0' then
                // Looping mode: Decrement the loop counter unless this is the
                // last iteration, in which case looping mode is exited.
                if !IsLastLowOverheadLoop(state) then
                    state.LoopCount = state.LoopCount - (1 << (4 - LTPSIZE))<31:0>;
                else
                    // LO_BRANCH_INFO.VALID does not need to be cleared at the end
                    // of the loop.
                    //
                    // Skip over LE at the end of the loop.
                    state.FetchAddr = state.FetchAddr + 4;
                    // Reset LTPSIZE if it's accessible, which will be the case
                    // for all predicated loops as the LETP instruction will
                    // have forced the state to be accessible and all operations
                    // that can cause the state to be inaccessible require a CSE
                    // which will invalidate LO_BRANCH_INFO.
```

```
                    if ActiveFPState() then
                        state.ResetLTPSize = TRUE;
    return state;
```

### E2.1.178 HasArchVersion

```
// HasArchVersion()
// ================

// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == Armv8p0 || boolean IMPLEMENTATION_DEFINED "Architecture version";
```

### E2.1.179 HaveDebugMonitor

```
// HaveDebugMonitor()
//===================

boolean HaveDebugMonitor()
    return HaveMainExt();
```

### E2.1.180 HaveDSPExt

```
// HaveDSPExt()
// ============
// Check whether DSP Extension is implemented.

boolean HaveDSPExt();
```

### E2.1.181 HaveDWT

```
// HaveDWT()
// =========
// Check whether Data Watchpoint and Trace unit is implemented.

boolean HaveDWT();
```

### E2.1.182 HaveFPB

```
// HaveFPB()
// =========
// Check whether Flash Patch and Breakpoint unit is implemented.

boolean HaveFPB();
```

### E2.1.183 HaveFPExt

```
// HaveFPExt()
// ============
// Check whether Floating Point Extension is implemented.

boolean HaveFPExt();
```

### E2.1.184 HaveHaltingDebug

```
// HaveHaltingDebug()
// ==================
// Check whether Halting debug implemented.

boolean HaveHaltingDebug();
```

### E2.1.185 HaveITM

```
// HaveITM()
// =========
// Check whether Instrumentation Trace Macrocell is implemented.

boolean HaveITM();
```

### E2.1.186 HaveLOBExt

```
// HaveLOBExt()
// ============
// Check whether the Low Overhead Loops and Branch Future Extension is implemented

boolean HaveLOBExt()
```

### E2.1.187 HaveMainExt

```
// HaveMainExt()
// =============
// Check whether Main Extension is implemented.

boolean HaveMainExt();
```

### E2.1.188 HaveMve

```
// HaveMve()
// =========
// Check whether M-profile Vector Extension is implemented

boolean HaveMve()
```

### E2.1.189 HaveMveOrFPExt

```
// HaveMveOrFPExt()
// ================

boolean HaveMveOrFPExt()
    return HaveFPExt() || HaveMve();
```

### E2.1.190 HaveSecurityExt

```
// HaveSecurityExt()
// =================
// Check whether the implementation have Security Extensions.

boolean HaveSecurityExt();
```

### E2.1.191 HaveSysTick

```
// HaveSysTick()
// =============
// Returns the number of SysTick instances (0, 1 or 2).

integer HaveSysTick();
```

### E2.1.192 HaveUDE

```
// HaveUDE()
// =========
// Check whether Unprivileged Debug Extension is implemented.

boolean HaveUDE()
```

### E2.1.193 HighestPri

```
// HighestPri()
// ============
// Priority of Thread mode with no active exceptions.

integer HighestPri()
    // The value is PriorityMax + 1 = 256 (configurable priority maximum bit field is 8 bits)
    return 256;
```

### E2.1.194 highestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

### E2.1.195 Hint_Debug

```
// Hint_Debug
// ==========
// Generate a hint to the debug system.

Hint_Debug(bits(4) option);
```

### E2.1.196 Hint_PreloadData

```
// Hint_PreloadData
// ================
// Performs a preload data hint

Hint_PreloadData(bits(32) address);
```

### E2.1.197 Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite
// ========================
// Performs a preload data hint for write.

Hint_PreloadDataForWrite(bits(32) address);
```

### E2.1.198 Hint_PreloadInstr

```
// Hint_PreloadInstr
// =================
// Performs a preload instructions hint

Hint_PreloadInstr(bits(32) address);
```

### E2.1.199 Hint_Yield

```
// Hint_Yield
// ==========
// Performs a Yield hint

Hint_Yield();
```

### E2.1.200 IDAUCheck

```
// IDAUCheck
// =========
// Query IDAU(Implementation Defined Attribution Unit) for attribution information

(boolean, boolean, boolean, bits(8), boolean) IDAUCheck(bits(32) address);
```

### E2.1.201 IgnoreFaultsType

```
// Indicates Ignore Faults Types
// =============================

enumeration IgnoreFaultsType { IgnoreFaults_NONE,
                               IgnoreFaults_STACK,
                               IgnoreFaults_ALL };
```

### E2.1.202 InITBlock

```
// InITBlock()
// ===========

boolean InITBlock(ITSTATEType itState)
    return (itState<3:0> != '0000');

boolean InITBlock()
    return InITBlock(ITSTATE);
```

### E2.1.203 InstrCanChain

```
// InstrCanChain()
// ===============

boolean InstrCanChain(bits(32) instr)
    // Check if the instruction is a chainable instruction, and if its a
    // chained memory operation.
    isChainMem = IsMveLoadStoreInstruction(instr);
    canChain   = IsMveBeatWiseInstruction(instr) && !InITBlock();

    // memory operations can't chain with other memory operations
    if canChain && isChainMem then
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if _InstInfo[i].Valid && IsMveLoadStoreInstruction(ThisInstr(i)) then
                canChain = FALSE;

    // Scalar dependencies must be tracked.
    if canChain then
        // Get a list of registers read and written by this instruction
        // (these are bitstring where the index of each set bit indicates a used register)
        myScalarReads  = GetMveScalarReadRegs(instr);
        myScalarWrites = GetMveScalarWriteRegs(instr);
        // Get a list of all registers read or written by other in-flight instructions
        otherScalarReads  = 0<15:0>;
        otherScalarWrites = 0<15:0>;
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
```

```
            if _InstInfo[i].Valid then
                otherScalarReads  = otherScalarReads  OR GetMveScalarReadRegs(_InstInfo[i].
                    Opcode);
                otherScalarWrites = otherScalarWrites OR GetMveScalarWriteRegs(_InstInfo[i].
                    Opcode);
        // Determine if there is any overlap between the registers read and written,
        // if so chaining is impossible.
        if ((myScalarReads AND otherScalarWrites) != 0<15:0>) ||
           ((myScalarWrites AND otherScalarReads) != 0<15:0>) then
            canChain = FALSE;

    // LR chaining restrictions
    if canChain && _InstInfo[0].Valid && LO_BRANCH_INFO.VALID == '1' && LO_BRANCH_INFO.BF ==
        '0' then
        // Check if any instruction in the chain writes to LR, and get the index
        // of the last instruction in the chain.
        lastValidId = 0;
        lrWrite     = FALSE;
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if _InstInfo[i].Valid then
                lastValidId = i;
                lrWrite = lrWrite || GetMveScalarWriteRegs(_InstInfo[i].Opcode)<14> == '1';
        // Don't chain the next instruction if the end of the loop body has been
        // reached and either one of the existing chained instructions writes to
        // LR, or the new instruction reads or writes to LR.
        instState   = GetInstrExecState(lastValidId);
        nextSeqAddr = instState.FetchAddr + _InstInfo[lastValidId].Length<31:0>;
        if LO_BRANCH_INFO.END_ADDR == nextSeqAddr<31:1> then
            canChain = !(lrWrite                                  ||
                         GetMveScalarReadRegs(instr)<14>  == '1' ||
                         GetMveScalarWriteRegs(instr)<14> == '1');

    // Two instructions reading/writing FPSCR carry bit should not chain with each other
    if canChain && IsMveAccessFPSCR_C(instr) then
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if _InstInfo[i].Valid && IsMveAccessFPSCR_C(_InstInfo[i].Opcode) then
                canChain = FALSE;

    // Branch future chaining restrictions
    if canChain && _InstInfo[0].Valid && LO_BRANCH_INFO.VALID == '1' && LO_BRANCH_INFO.BF ==
        '1' then
        // Get the index of the last instruction in the chain.
        lastValidId = 0;
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if _InstInfo[i].Valid then
                lastValidId = i;
        // Don't chain the next instruction if execution has reached a BF branch
        // point
        instState   = GetInstrExecState(lastValidId);
        nextSeqAddr = instState.FetchAddr + _InstInfo[lastValidId].Length<31:0>;
        canChain    = LO_BRANCH_INFO.END_ADDR != nextSeqAddr<31:1>;

    // Implementations can choose not to chain an instruction
    if canChain then
        canChain = boolean IMPLEMENTATION_DEFINED "Chain instruction";

    return canChain;
```

## E2.1.204 InstrExecState

```
// Indicates instruction execution state
// =====================================

type INSTR_EXEC_STATE_Type is (
    bits(32)    FetchAddr,
    ITSTATEType ITState,
    bit         L,
    bit         T16IND,
```

```
    bits(32)    LoopCount,
    bit         LOBranchInfoValid,
    boolean     ResetLTPSize
)
INSTR_EXEC_STATE_Type _CurrentInstrExecState;
```

## E2.1.205 InstructionAdvance

```
// InstructionAdvance()
// ====================

InstructionAdvance(boolean commitState)
    if _PCChanged then
        // We can still accept branches (eg: if an exception occurs) even
        // if the current instruction is not yet ready to be committed. This
        // is combined with a different return value from ReturnState to always
        // point towards the correct return address.
        _RName[RNamesPC] = NextInstrAddr();
        if HaveMainExt() then
            EPSR.IT  = NextInstrITState();
    if commitState then
        // Instruction getting old (or scalar instruction).
        // Commit next state back to the registers.
        INSTR_EXEC_STATE_Type next = GetInstrExecState(1);
        if HaveLOBExt() then
            if next.LOBranchInfoValid == '1' then
                LR = next.LoopCount;
            else
                LO_BRANCH_INFO.VALID = '0';
            if next.L == '1' then
                // Set LR to return to the return address. The offset of the
                // return address depends on whether the originating BF
                // instruction assumed there would be a T32 or a T16
                // instruction after the branch point. See BF documentation for
                // details.
                retAddr = ThisInstrAddr() + ThisInstrLength();
                retAddr = retAddr + (if next.T16IND == '1' then 2 else 4);
                LR      = retAddr<31:1> : '1';
            if HaveMve() && next.ResetLTPSize then
                FPSCR.LTPSIZE = 4<2:0>;
        _RName[RNamesPC] = next.FetchAddr;
        if HaveMainExt() then
            EPSR.IT  = next.ITState;

        // Mark an instruction as having retired
        DHCSR.S_RETIRE_ST = '1';

        // Advance the instruction FIFO
        for i = 0 to MAX_OVERLAPPING_INSTRS-1
            if i == MAX_OVERLAPPING_INSTRS-1 then
                _InstInfo[i].Valid  = FALSE;
            else
                _InstInfo[i] = _InstInfo[i+1];

    _ITStateChanged = FALSE;
    _PCChanged = FALSE;
```

## E2.1.206 InstructionExecute

```
// InstructionExecute()
// ====================

// If fetchNew is set then fetch and execute new instructions, otherwise only
// continue execution of inflight beats.
boolean InstructionExecute(boolean fetchNew)
    try
        // Attempt to execute the next instruction. Start by setting up the state.
```

```
        _InstID                = 0;
        _BeatID                = 0;
        activeChains           = GetActiveChains();
        _CurrentInstrExecState = GetInstrExecState(activeChains);
        commitState           = FALSE;
        // Fetch the instruction
        pc                     = ThisInstrAddr();
        (instr, is16bit)       = FetchInstr(pc);
        len                    = if is16bit then 2 else 4;

        // Checking for FPB Breakpoint on instructions
        if HaveFPB() && FPB_CheckBreakPoint(pc, len, TRUE, IsSecure()) then
            GenerateDebugEventResponse(TRUE);

        // If a chain is being executed then the current instruction can be
        // added to the chain if it's chainable. If the instruction is not
        // chainable (eg because its a scaler instruction) then it is not
        // executed, and the next beat(s) of the flight chained instructions
        // are executed. This process is repeated on the next architecture
        // tick, and when the chain has completed the non-chainable
        // instruction can be executed.
        // NOTE: Chainable instructions aren't allowed to chain if in an IT
        //       block.
        chainableInst = IsMveBeatWiseInstruction(instr) && !InITBlock();
        if HaveMve() && (chainableInst || activeChains > 0) then
            // A new instruction can only be chained if a power of 2 number
            // of beats have completed. Also allow an instruction to be
            // started if the ECI information indicates that an instruction
            // is in progress, but the corresponding slot in the instruction
            // queue is empty, which can occur on exception return.
            chainableExecPoint = (EPSR.ECI == 0<7:0>) || (EPSR.ECI == 2<7:0>);
            if !chainableExecPoint then
                beatStatus = BeatComplete;
                for instId = 0 to MAX_OVERLAPPING_INSTRS-1
                    if ((Elem[beatStatus, instId, MAX_BEATS] != Zeros(MAX_BEATS)) &&
                        !_InstInfo[instId].Valid) then
                        chainableExecPoint = TRUE;
            if chainableExecPoint && InstrCanChain(instr) && fetchNew then
                SetThisInstrDetails(instr, len);
            commitState = ExecBeats();
        elsif fetchNew then
            // Scalar instruction, execute instructions normally.
            SetThisInstrDetails(instr, len);
            InstStateCheck(instr);
            DecodeExecute(instr, pc, is16bit, DefaultCond());
            // Scalar instructions, and MVE instructions in IT blocks don't
            // have beat behaviour so commit straight away
            commitState = TRUE;

        // Check for DWT match
        if IsDWTEnabled() then DWT_InstructionMatch(pc);

    catch exn
        // Do not catch UNPREDICTABLE or internal errors
        when IsSEE(exn) || IsUNDEFINED(exn)
            // Unallocated instructions in the NOP hint space and instructions
            // that fail their condition tests are treated like NOP's.
            nopHint = instr IN {'00000000000000000010111111xxxx0000',
                                '11110011101011111000000xxxxxxxx'};
            if ConditionHolds(CurrentCond()) && !nopHint then
                commitState = FALSE;
                toSecure    = IsSecure();
                // Unallocated instructions in the coprocessor space behave as NOCP
                // if the coprocessor is disabled.
                (isCp, cpNum) = IsCPInstruction(instr);
                if isCp then
                    (cpEnabled, cpFaultState) = IsCPEnabled(cpNum);
                if isCp && !cpEnabled then
                    // A PE is permitted to decode the coprocessor space and raise
```

```
                    // UNDEFINSTR UsageFaults for unallocated encodings even if the
                    // coprocessor is disabled.
                    if boolean IMPLEMENTATION_DEFINED "Decode CP space" then
                        UFSR.UNDEFINSTR = '1';
                    else
                        UFSR.NOCP       = '1';
                        toSecure        = cpFaultState;
                else
                    UFSR.UNDEFINSTR = '1';

                // If Main Extension is not implemented the fault will escalate
                //  to a HardFault.
                excInfo = CreateException(UsageFault, TRUE, toSecure);
                // Prevent EndOfInstruction() being called in
                // HandleException() as the instruction has already been
                // terminated so there is no need to throw the exception
                // again.
                excInfo.termInst = FALSE;
                HandleException(excInfo);
            else
                // If the instruction condition does not pass then this
                // behaves as a NOP, as such PC must be advanced. Since
                // vector instructions are not chained (thus only one
                // instruction is in flight) when inside an IT
                // block, they are also committed here.
                commitState = TRUE;
        when IsExceptionTaken(exn)
            // If an exception is thrown then it was before commitState was
            // set to true, so no additional actions are required in this
            // catch block.

    return commitState;
```

### E2.1.207 InstructionsInFlight

```
// InstructionsInFlight()
// ======================

boolean InstructionsInFlight()
  // If there is more than one active chain and it isn't just a single active
  // scalar instruction then there are instructions in flight
  return GetActiveChains() != 0 && (GetActiveChains() != 1 ||
                                    IsMveBeatWiseInstruction(_InstInfo[0].Opcode));
```

### E2.1.208 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================
// Perform an instruction synchronization barrier operation

InstructionSynchronizationBarrier(bits(4) option);
```

### E2.1.209 InstStateCheck

```
// InstStateCheck()
// ================

InstStateCheck(bits(32) instr)
    // Check for IT,ICI,ECI bits that are not permitted for the current
    // instruction. NOTE EPSR.ICI and EPSR.ECI overlap with EPSR.IT.
    validICI = EPSR.ICI<7:4> == Zeros(4);
    validECI = UInt(EPSR.ECI) < 6 && EPSR.ECI<3:0> != '0011';
    valid    = ( InITBlock()                                          ||
                 EPSR.IT == Zeros(8)                                  ||
                 (validICI && (IsLoadStoreClearMultInstruction(instr) ||
```

```
                                  IsBKPTInstruction(instr)))             ||
                  (validECI && (IsMveBeatWiseInstruction(instr)          ||
                                  IsLEInstruction(instr)                 ||
                                  IsBKPTInstruction(instr))) );
    if !valid then
        UFSR.INVSTATE = '1';
        excInfo = CreateException(UsageFault);
        HandleException(excInfo);
```

### E2.1.210 Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

### E2.1.211 IntegerZeroDivideTrappingEnabled

```
// IntegerZeroDivideTrappingEnabled()
// ==================================

boolean IntegerZeroDivideTrappingEnabled()
    // DIV_0_TRP bit in CCR is RAZ/WI if Main Extension is not implemented
    return CCR.DIV_0_TRP == '1';
```

### E2.1.212 InvalidateFPRegs

```
// InvalidateFPRegs()
// ==================

InvalidateFPRegs(boolean shouldClear, boolean doCallee)
    clearValue = if shouldClear then Zeros(32) else bits(32) UNKNOWN;

    for i = 0 to 15
        S[i]                    = clearValue;
        if doCallee then S[i+16] = clearValue;
    FPSCR = clearValue;
    VPR   = clearValue;
```

### E2.1.213 IsAccessible

```
// IsAccessible()
// ==============

(bit, bit, bits(8), boolean) IsAccessible(bits(32) address, boolean forceunpriv,
                                          boolean isSecure)
    bit write;
    bit read;

    // Work out which privilege level the current mode in the Non-secure state
    // is subject to
    if forceunpriv then
        isPrivileged = FALSE;
    else
        isPrivileged = CurrentModeIsPrivileged(isSecure);
    (-, perms) = MPUCheck(address, AccType_NORMAL, isPrivileged, isSecure);
    if !perms.apValid then
        write = '0';
        read  = '0';
    else
        case perms.ap of
            when '00'  (write, read) = if isPrivileged then ('1','1') else ('0','0');
```

```
            when '01'  (write, read) = ('1','1') ;
            when '10'  (write, read) = if isPrivileged then ('0','1') else ('0','0');
            when '11'  (write, read) = ('0','1');
    return (write, read, perms.region, perms.regionValid);
```

### E2.1.214 IsActiveForState

```
// IsActiveForState()
// ==================

boolean IsActiveForState(integer exception, boolean isSecure)
    if !HaveSecurityExt() then
        isSecure = FALSE;
    // If the exception is configurable then check which domain it
    // currently targets. If its not configurable then the active flags can be
    // used directly.
    if IsExceptionTargetConfigurable(exception) then
        active = ((ExceptionActive[exception] != '00') &&
                  (ExceptionTargetsSecure(exception, isSecure) == isSecure));
    else
        idx    = if isSecure then 0 else 1;
        active = ExceptionActive[exception]<idx> == '1';
    return active;
```

### E2.1.215 IsAligned

```
// IsAligned()
// ===========

boolean IsAligned(bits(32) address, integer size)
    assert size IN {1,2,4,8};
    mask = (size-1)<31:0>;                  // integer to bit string conversion
    return IsZero(address AND mask);
```

### E2.1.216 IsBKPTInstruction

```
// IsBKPTInstruction()
// ===================
// Checks whether the instruction is a breakpoint

boolean IsBKPTInstruction(bits(32) instr)
    return instr == '0000 0000 0000 0000  1011 1110 xxxxxxxx';
```

### E2.1.217 IsCPEnabled

```
// IsCPEnabled()
// =================

(boolean, boolean) IsCPEnabled(integer cp, boolean privileged, boolean secure)
    // Check Coprocessor Access Control Register for permission to use coprocessor
    boolean enabled;
    boolean forceToSecure = FALSE;

    cpacr = if secure then CPACR_S else CPACR_NS;
    case cpacr<(cp*2)+1:cp*2> of
        when '00'
            enabled = FALSE;
        when '01'
            enabled = privileged;
        when '10'
            UNPREDICTABLE;
        when '11' // access permitted by CPACR
            enabled = TRUE;

```

```
    if enabled && HaveSecurityExt() then
        // Check if access in forbidden by NSACR
        if !secure && NSACR<cp> == '0' then
            enabled       = FALSE;
            forceToSecure = TRUE;

    // Check if the coprocessor state unknown flag.
    if enabled && CPPWR_S<cp*2> == '1' then
        enabled = FALSE;
        // Check SUS bit to determine the target state of any fault
        forceToSecure = CPPWR_S<(cp*2)+1> == '1';

    return (enabled, secure || forceToSecure);

(boolean, boolean) IsCPEnabled(integer cp)
    return IsCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());
```

## E2.1.218 IsCPEnabled

```
// IsCPEnabled()
// =============

(boolean, boolean) IsCPEnabled(integer cp, boolean privileged, boolean secure)
    // Check Coprocessor Access Control Register for permission to use coprocessor
    boolean enabled;
    boolean forceToSecure = FALSE;

    cpacr = if secure then CPACR_S else CPACR_NS;
    case cpacr<(cp*2)+1:cp*2> of
        when '00'
            enabled = FALSE;
        when '01'
            enabled = privileged;
        when '10'
            UNPREDICTABLE;
        when '11' // access permitted by CPACR
            enabled = TRUE;

    if enabled && HaveSecurityExt() then
        // Check if access in forbidden by NSACR
        if !secure && NSACR<cp> == '0' then
            enabled       = FALSE;
            forceToSecure = TRUE;

    // Check if the coprocessor state unknown flag.
    if enabled && CPPWR_S<cp*2> == '1' then
        enabled = FALSE;
        // Check SUS bit to determine the target state of any fault
        forceToSecure = CPPWR_S<(cp*2)+1> == '1';

    return (enabled, secure || forceToSecure);

(boolean, boolean) IsCPEnabled(integer cp)
    return IsCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());
```

## E2.1.219 IsCPInstruction

```
// IsCPInstruction()
// =================

(boolean, integer) IsCPInstruction(bits(32) instr)
    isCp  = instr IN { '111x1110xxxxxxxxxxxxxxxxxxxxxxxx',
                       '111x110xxxxxxxxxxxxxxxxxxxxxxxx' };
    cpNum = if isCp then UInt(instr<11:8>) else integer UNKNOWN;
    // CP 11 controlled by CP10 enables.
    // As of v8.1 CP8, 9, 14 and 15 are also controlled by CP10 enables.
    if (  cpNum IN {11} ||
```

```
        (cpNum IN {8, 9, 14, 15} && HasArchVersion(Armv8p1)) ) then
        cpNum = 10;
    // From v8.1 the encoding space outside the CDP space used by MVE instructions
    // is also classed as coprocessor space and is associated with CP10.
    if instr IN { '111x1111xxxxxxxxxxxxxxxxxxxxxxxx'} && HasArchVersion(Armv8p1) then
        isCp  = TRUE;
        cpNum = 10;
    return (isCp, cpNum);
```

## E2.1.220 IsDebugState

```
// IsDebugState
// =============

boolean IsDebugState()
    return Halted;
```

## E2.1.221 IsDWTConfigUnpredictable

```
// IsDWTConfigUnpredictable()
// ==========================
// Checks for the UNPREDICTABLE cases for various combination of MATCH and
// ACTION for each comparator.

boolean IsDWTConfigUnpredictable(integer N)

    no_trace = (!HaveMainExt() || DWT_CTRL.NOTRCPKT == '1' || !HaveITM());

    // First pass check of MATCH field - coarse checks
    case DWT_FUNCTION[N].MATCH of
        when '0000'                             // Disabled
            return FALSE;
        when '0001'                             // Cycle counter match
            if !HaveMainExt() || DWT_CTRL.NOCYCCNT == '1' || DWT_FUNCTION[N].ID<0> == '0'
                then
                return TRUE;
        when '001x'                             // Instruction address
            if (DWT_FUNCTION[N].ID<1> == '0' || DWT_FUNCTION[N].DATAVSIZE != '01' ||
                DWT_COMP[N]<0> == '1') then
                return TRUE;
        when '01xx'                             // Data address
            lsb = UInt(DWT_FUNCTION[N].DATAVSIZE);
            if DWT_FUNCTION[N].ID<3> == '0' || (lsb > 0 && !IsZero(DWT_COMP[N]<lsb-1:0>))
                then
                return TRUE;
        when '1100', '1101', '1110'             // Data address with value
            if no_trace then return TRUE;
            lsb = UInt(DWT_FUNCTION[N].DATAVSIZE);
            if DWT_FUNCTION[N].ID<3> == '0' || (lsb > 0 && !IsZero(DWT_COMP[N]<lsb-1:0>))
                then
                return TRUE;
        when '10xx'                             // Data value
            Vsize = 2^UInt(DWT_FUNCTION[N].DATAVSIZE);
            if (!HaveMainExt() || DWT_FUNCTION[N].ID<2> == '0' ||
                (Vsize != 4 && DWT_COMP[N]<31:16> != DWT_COMP[N]<15:0>) ||
                (Vsize == 1 && DWT_COMP[N]<15:8> != DWT_COMP[N]<7:0>)) then
                return TRUE;
            if (HasArchVersion(Armv8p1) &&
                (!IsZero(DWT_VMASK[N] AND DWT_COMP[N]) ||
                 (Vsize != 4 && DWT_VMASK[N]<31:16> != DWT_VMASK[N]<15:0>) ||
                 (Vsize == 1 && DWT_VMASK[N]<15:8> != DWT_VMASK[N]<7:0>))) then
                return TRUE;
        otherwise
            return TRUE;

    // Second pass MATCH check - linked and limit comparators
    case DWT_FUNCTION[N].MATCH of
```

```
        when '0011'                             // Instruction address limit
            if (N == 0 || DWT_FUNCTION[N].ID<4> == '0' ||
                DWT_FUNCTION[N-1].MATCH IN {'0001','0011','01xx','1xxx'} ||
                UInt(DWT_COMP[N]) <= UInt(DWT_COMP[N-1])) then
                return TRUE;
            if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
        when '0111'                             // Data address limit
            if (N == 0 || DWT_FUNCTION[N].ID<4> == '0' ||
                DWT_FUNCTION[N-1].MATCH IN {'0001','001x','0111','10xx'} ||
                DWT_FUNCTION[N].DATAVSIZE != '00' || DWT_FUNCTION[N-1].DATAVSIZE != '00' ||
                UInt(DWT_COMP[N]) <= UInt(DWT_COMP[N-1])) then
                return TRUE;
            if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
        when '1011'                             // Linked data value
            if (N == 0 || DWT_FUNCTION[N].ID<4> == '0' ||
                DWT_FUNCTION[N-1].MATCH IN {'0001','001x','0111','10xx'} ||
                DWT_FUNCTION[N].DATAVSIZE != DWT_FUNCTION[N-1].DATAVSIZE) then
                return TRUE;
            if DWT_FUNCTION[N-1].MATCH == '0000' then return FALSE;
        otherwise
            // No limitations in second pass

    // Check DATAVSIZE is permitted
    if DWT_FUNCTION[N].DATAVSIZE == '11' then return TRUE;

    // Check the ACTION is allowed for the MATCH type
    case DWT_FUNCTION[N].ACTION of
        when '00'                               // CMPMATCH trigger only
            if DWT_FUNCTION[N].MATCH IN {'1100', '1101', '1110'} then
                return TRUE;
        when '01'                               // Debug event
            if DWT_FUNCTION[N].MATCH IN {'0011', '0111', '1100', '1101', '1110'} then
                return TRUE;
        when '10'                               // Data Trace Match or Data Value packet
            if no_trace || DWT_FUNCTION[N].MATCH IN {'0011', '0111'} then
                return TRUE;
        when '11'                               // Other Data Trace packet
            if (no_trace || DWT_FUNCTION[N].MATCH IN {'0010', '1000', '1001', '1010'} ||
                (DWT_FUNCTION[N].MATCH == '0011' && DWT_FUNCTION[N-1].ACTION != '00') ||
                (DWT_FUNCTION[N].MATCH == '0111' && DWT_FUNCTION[N-1].MATCH == '01xx' &&
                 DWT_FUNCTION[N-1].ACTION IN {'01', '10'}) ||
                (DWT_FUNCTION[N].MATCH == '0111' && DWT_FUNCTION[N-1].MATCH == '11xx' &&
                 DWT_FUNCTION[N-1].ACTION IN {'00', '01'})) then
               return TRUE;

    return FALSE;                               // Passes checks
```

## E2.1.222 IsDWTEnabled

```
// IsDWTEnabled()
// ==============
// Check whether DWT is enabled.

boolean IsDWTEnabled()
   return HaveDWT() && DEMCR.TRCENA == '1' && NoninvasiveDebugAllowed();
```

## E2.1.223 IsExceptionTargetConfigurable

```
// IsExceptionTargetConfigurable()
// ===============================

boolean IsExceptionTargetConfigurable(integer e)
    if HaveSecurityExt() then
        case e of
            when NMI
                configurable = TRUE;
            when BusFault
```

```
                configurable = TRUE;
            when DebugMonitor
                configurable = TRUE;
            when SysTick
                // If there is only 1 SysTick instance then the target domain is
                // configurable.
                configurable = HaveSysTick() == 1;
            otherwise
                // Exceptions numbers lower than 16 that are not listed in this
                // function are not configurable in this context.
                configurable = e >= 16;
    else
        configurable = FALSE;
    return configurable;
```

### E2.1.224 IsExclusiveGlobal

```
// IsExclusiveGlobal
// =================
// Checks if PE has marked in a global record an address range as "exclusive access
// requested" that covers at least the size bytes from address

boolean IsExclusiveGlobal(bits(32) address, integer processorid, integer size);
```

### E2.1.225 IsExclusiveLocal

```
// IsExclusiveLocal
// ================
// Checks if PE has marked in a local record an address range as "exclusive access
// requested" that covers at least the size bytes from address

boolean IsExclusiveLocal(bits(32) address, integer processorid, integer size);
```

### E2.1.226 IsFirstBeat

```
// IsFirstBeat()
// =============

boolean IsFirstBeat()
    return _BeatID == 0;
```

### E2.1.227 IsIrqValid

```
// IsIrqValid()
// ============
// Check whether given exception number denotes a valid external interrupt
// implemented by PE.

boolean IsIrqValid(integer e);
```

### E2.1.228 IsLastBeat

```
// IsLastBeat()
// ============

boolean IsLastBeat()
    return _BeatID >= (MAX_BEATS - 1);
```

### E2.1.229 IsLastLowOverheadLoop

```
// IsLastLowOverheadLoop()
// =======================

boolean IsLastLowOverheadLoop()
    return IsLastLowOverheadLoop(_CurrentInstrExecState);

boolean IsLastLowOverheadLoop(INSTR_EXEC_STATE_Type state)
    // This does not check whether a loop is currently active.
    // If we were in a loop, would this be the last one?
    return UInt(state.LoopCount) <= (1 << (4 - LTPSIZE));
```

### E2.1.230 IsLEInstruction

```
// IsLEInstruction()
// =================
// Checks whether the instruction is a loop end instruction

boolean IsLEInstruction(bits(32) instr);
```

### E2.1.231 IsLoadStoreClearMultInstruction

```
// IsLoadStoreClearMultInstruction()
// =================================
// Checks whether the instruction is a clear multiple or a load / store multiple

boolean IsLoadStoreClearMultInstruction(bits(32) instr);
```

### E2.1.232 isOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

### E2.1.233 IsReqExcPriNeg

```
// IsReqExcPriNeg()
// ================

boolean IsReqExcPriNeg(boolean secure)
    // This function checks if the requested execution priority is negative for
    // the specified security domain. That is, NMI or HardFault is active, or
    // FAULTMASK is set. It does not take account of AIRCR.PRIS so returns TRUE
    // if FAULTMASK_NS is set even if PRIS is set to restrict Non-secure priorities
    // to the range 0x80-0x7E
    neg = IsActiveForState(NMI, secure) || IsActiveForState(HardFault, secure);
    if HaveMainExt() then
        faultmask = if secure then FAULTMASK_S else FAULTMASK_NS;
        if faultmask.FM == '1' then
            neg = TRUE;
    return neg;


boolean IsReqExcPriNeg(boolean secure, AccType acctype)
    // If the access is due to lazy FP state preservation the FPCCR flag
    // indicating whether a HardFault could be taken is used to determine if the
    // priority should be considered to be negative rather than the current
    // execution priority.
    if acctype == AccType_LAZYFP then
        neg = FPCCR_S.HFRDY == '0';
    else
        neg = IsReqExcPriNeg(secure);
    return neg;
```

### E2.1.234 IsReturn

```
// IsReturn()
// ==========

AddrType IsReturn(bits(32) address)
    addrtype = AddrType_NORMAL;

    if (HaveSecurityExt() && address=='1111 1110 1111 1111 1111 1111 1111 111x') then
        addrtype = AddrType_FNC_RETURN;

    elsif CurrentMode() == PEMode_Handler && address<31:24> == '11111111' then
        addrtype = AddrType_EXC_RETURN;
    return addrtype;
```

### E2.1.235 IsSecure

```
// IsSecure()
// ==========

boolean IsSecure()
    return HaveSecurityExt() && CurrentState == SecurityState_Secure;
```

### E2.1.236 isZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

### E2.1.237 isZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

### E2.1.238 ITAdvance

```
// ITAdvance()
// ===========

ITSTATEType ITAdvance(ITSTATEType itState)
    // If the mask field (I.E. the bottom 4 bits) are zero then the ITSTATE bits
    // hold ECI information and therefore the normal state advancement should
    // not take place.
    if itState<3:0> == '1000' then
        itState = '00000000';
    elsif itState<3:0> != '0000' then
        itState<4:0> = LSL(itState<4:0>, 1);
    return itState;
```

### E2.1.239 ITSTATE

```
// ITSTATE
// =======

ITSTATEType ITSTATE
    return ThisInstrITState();

```

```
ITSTATE = ITSTATEType value
    // Writes to ITSTATE don't take effect immediately, instead they change the
    // value returned by NextInstrITState().
    _NextInstrITState = value;
    _ITStateChanged   = TRUE;
```

### E2.1.240 ITSTATEType

```
// If-Then execution state bits for the T32 IT instruction.

type ITSTATEType = bits(8);
```

### E2.1.241 LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE<3:0> == '1000');
```

### E2.1.242 LoadWritePC

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

LoadWritePC(bits(32) address, integer baseReg, bits(32) baseRegVal, boolean baseRegUpdate,
            boolean spLimCheck)

    if baseRegUpdate then
        oldBaseVal = R[baseReg];
        if spLimCheck then
            RSPCheck[baseReg] = baseRegVal;
        else
            R[baseReg]        = baseRegVal;

    // Attempt to update the PC, which may result in a fault
    excInfo = BranchReturn(address, FALSE);

    if baseRegUpdate && excInfo.fault != NoFault then
        // Restore the previous base reg value, SP limit checking is not performed
        if baseReg == 13 then
            exc = _SP(LookUpRName(baseReg), FALSE, TRUE, oldBaseVal);
            assert exc.fault == NoFault;
        else
            R[baseReg] = oldBaseVal;

    HandleException(excInfo);
```

### E2.1.243 LockedUp

```
// Indicates the PE is locked up

boolean LockedUp;
```

### E2.1.244 Lockup

```
// Lockup()
// ========

Lockup(boolean termInst)
    LockedUp = TRUE;
    // Branch to the lockup address.
    BranchTo(0xEFFFFFFE<31:0>, TRUE);
```

```
    // Invalidate the instruction buffer and set the length of the current
    // instruction to zero so NextInstrAddr() reports the correct lockup
    // address.
    for i = 0 to MAX_OVERLAPPING_INSTRS-1
        _InstInfo[i].Valid = FALSE;
    _InstInfo[0].Length = 0;
    // If requested, terminate execution of the pseudo code for this
    // instruction.
    if termInst then
        EndOfInstruction();
```

### E2.1.245 LookUpRName

```
// LookUpRName()
// =============

RNames LookUpRName(integer n)
    case n of
        when 0  result = RNames0;
        when 1  result = RNames1;
        when 2  result = RNames2;
        when 3  result = RNames3;
        when 4  result = RNames4;
        when 5  result = RNames5;
        when 6  result = RNames6;
        when 7  result = RNames7;
        when 8  result = RNames8;
        when 9  result = RNames9;
        when 10 result = RNames10;
        when 11 result = RNames11;
        when 12 result = RNames12;
        when 13 result = LookUpSP();
        when 14 result = RNamesLR;
        when 15 result = RNamesPC;
        otherwise assert(FALSE);
    return result;
```

### E2.1.246 LookUpSP

```
// LookUpSP()
// ==========

RNames LookUpSP()
    return LookUpSP_with_security_mode(IsSecure(), CurrentMode());
```

### E2.1.247 LookUpSP_with_security_mode

```
// LookUpSP_with_security_mode()
// =============================

RNames LookUpSP_with_security_mode(boolean isSecure, PEMode mode)
    RNames sp;
    bit spSel;

    // Get the SPSEL bit corresponding to the Security state requested
    if isSecure then
        spSel = CONTROL_S.SPSEL;
    else
        spSel = CONTROL_NS.SPSEL;

    // Should we be using the process or main stack pointers
    if spSel == '1' && mode == PEMode_Thread then
        if isSecure then
            sp = RNamesSP_Process_Secure;
        else
```

```
            sp = RNamesSP_Process_NonSecure;
    else
        if isSecure then
            sp = RNamesSP_Main_Secure;
        else
            sp = RNamesSP_Main_NonSecure;
    return sp;
```

### E2.1.248 LookUpSPLim

```
// LookUpSPLim()
// =============

bits(32) LookUpSPLim(RNames spreg)
    case spreg of
        when RNamesSP_Main_Secure    limit = MSPLIM_S.LIMIT:'000';
        when RNamesSP_Process_Secure limit = PSPLIM_S.LIMIT:'000';
        when RNamesSP_Main_NonSecure
            limit = if HaveMainExt() then MSPLIM_NS.LIMIT:'000' else Zeros(32);
        when RNamesSP_Process_NonSecure
            limit = if HaveMainExt() then PSPLIM_NS.LIMIT:'000' else Zeros(32);
        otherwise
            assert (FALSE);

    return limit;
```

### E2.1.249 lowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

### E2.1.250 LR

```
// LR
// ==

// Non-assignment form
bits(32) LR
    return RName[RNamesLR];

// Assignment form

LR = bits(32) value
    RName[RNamesLR] = value;
```

### E2.1.251 LSL

```
// 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;
```

### E2.1.252 LSL_C

```
// 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);

(bits(N), bits(M)) LSL_C(bits(N) x, bits(M) carry_in, integer shift)
    assert shift > 0 && shift <= M;
    cin        = LSL(carry_in,  M - shift);
    extended_x = LSL(Zeros(M) : x : cin, shift);
    result     = extended_x<N+  M-1:M  >;
    carry_out  = extended_x<N+2*M-1:N+M>;
    return (result, carry_out);
```

### E2.1.253 LSR

```
// 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;
```

### E2.1.254 LSR_C

```
// 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);
```

### E2.1.255 LTPSIZE

```
// LTPSIZE - non-assignment form
// =============================

integer LTPSIZE
    if HaveMve() && ActiveFPState() then
        size = UInt(FPSCR.LTPSIZE);
    else
        // Full vector length, so no loop tail predication
        size = 4;
    return size;
```

### E2.1.256 MAIRDecode

```
// MAIRDecode()
// ============

```

```
MemoryAttributes MAIRDecode(bits(8) attrfield, bits(2) sh)
    // Converts the MAIR attributes to orthogonal attribute and
    // hint fields.
    MemoryAttributes memattrs;
    // Decoding MAIR0/MAIR1 Registers
    if attrfield<7:4> == '0000' then
        unpackinner = FALSE;
        memattrs.memtype = MemType_Device;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
        memattrs.innerattrs = bits(2) UNKNOWN;
        memattrs.outerattrs = bits(2) UNKNOWN;
        memattrs.innerhints = bits(2) UNKNOWN;
        memattrs.outerhints = bits(2) UNKNOWN;
        memattrs.innertransient = boolean UNKNOWN;
        memattrs.outertransient = boolean UNKNOWN;
        case attrfield<3:0> of
            when '0000' memattrs.device = DeviceType_nGnRnE;
            when '0100' memattrs.device = DeviceType_nGnRE;
            when '1000' memattrs.device = DeviceType_nGRE;
            when '1100' memattrs.device = DeviceType_GRE;
            otherwise UNPREDICTABLE;
    else
        unpackinner = TRUE;
        memattrs.memtype = MemType_Normal;
        memattrs.device = DeviceType UNKNOWN;
        memattrs.outerhints = attrfield<5:4>;
        memattrs.shareable = sh<1> == '1';
        memattrs.outershareable = sh == '10';
        if sh == '01' then UNPREDICTABLE;

        if attrfield<7:6> =='00' then
            memattrs.outerattrs = '10';
            memattrs.outertransient = TRUE;
        elsif attrfield<7:6> =='01' then
            if attrfield<5:4> == '00' then
                memattrs.outerattrs = '00';
                memattrs.outertransient = FALSE;
            else
                memattrs.outerattrs = '11';
                memattrs.outertransient =  TRUE;
        else
            memattrs.outerattrs = attrfield<7:6>;
            memattrs.outertransient = FALSE;
    if unpackinner then
        if attrfield<3:0> == '0000' then UNPREDICTABLE;
        else
            if attrfield<3:2> =='00' then
                memattrs.innerattrs   = '10';
                memattrs.innerhints   = attrfield<1:0>;
                memattrs.innertransient = TRUE;
            elsif attrfield<3:2> =='01' then
                memattrs.innerhints   = attrfield<1:0>;
                if attrfield<1:0> == '00' then
                    memattrs.innerattrs = '00';
                    memattrs.innertransient = FALSE;
                else
                    memattrs.innerattrs = '11';
                    memattrs.innertransient =  TRUE;
            elsif attrfield<3:2> =='10' then
                    memattrs.innerhints   = attrfield<1:0>;
                    memattrs.innerattrs   = '10';
                    memattrs.innertransient = FALSE;
            elsif attrfield<3:2> =='11' then
                    memattrs.innerhints   = attrfield<1:0>;
                    memattrs.innerattrs   = '11';
                    memattrs.innertransient = FALSE;
            else  UNPREDICTABLE;
    return memattrs;
```

### E2.1.257 MarkExclusiveGlobal

```
// MarkExclusiveGlobal
// ===================
// Records in a global record that PE has requested "exclusive access" covering
// at least size bytes from the address

MarkExclusiveGlobal(bits(32) address, integer processorid, integer size);
```

### E2.1.258 MarkExclusiveLocal

```
// MarkExclusiveLocal
// ==================
// Records in a local record that PE has requested "exclusive access" covering
// at least size bytes from the address.

MarkExclusiveLocal(bits(32) address, integer processorid, integer size);
```

### E2.1.259 max

```
// Max()
// =====

__overloaded integer Max(integer a, integer b)
    return if a >= b then a else b;

__overloaded real Max(real a, real b)
    return if a >= b then a else b;
```

### E2.1.260 MaxExceptionNum

```
// MaxExceptionNum()
// =================
// Returns the maximum exception number supported

integer MaxExceptionNum()
    if HaveMainExt() then
        return 511;
    else
        return 47;
```

### E2.1.261 MemA

```
// MemA[]
// ======

bits(8*size) MemA[bits(32) address, integer size]
    return MemA_with_priv[address, size, FindPriv(), TRUE];

MemA[bits(32) address, integer size] = bits(8*size) value
    MemA_with_priv[address, size, FindPriv(), TRUE] = value;
    return;
```

### E2.1.262 MemA_MVE

```
// MemA_MVE[]
// ==========

// Non-assignment form

bits(8*size) MemA_MVE[bits(32) address, integer size]
    (excInfo, value) = MemA_with_priv_security(address, size, AccType_MVE,
```

```
                                                FindPriv(), IsSecure(), TRUE);
    HandleException(excInfo);
    return value;


// Assignment form

MemA_MVE[bits(32) address, integer size] = bits(8*size) value
    excInfo = MemA_with_priv_security(address, size, AccType_MVE, FindPriv(),
                                      IsSecure(), TRUE, value);
    HandleException(excInfo);
```

### E2.1.263 MemA_with_priv

```
// MemA_with_priv[]
// ================

// Non-assignment form

bits(8*size) MemA_with_priv[bits(32) address, integer size, boolean privileged,
                            boolean aligned]
    (excInfo, value) = MemA_with_priv_security(address, size, AccType_NORMAL,
                                               privileged, IsSecure(), aligned);
    HandleException(excInfo);
    return value;


// Assignment form

MemA_with_priv[bits(32) address, integer size, boolean privileged,
               boolean aligned] = bits(8*size) value
    excInfo = MemA_with_priv_security(address, size, AccType_NORMAL, privileged,
                                      IsSecure(), aligned, value);
    HandleException(excInfo);
```

### E2.1.264 MemA_with_priv_security

```
// MemA_with_priv_security()
// =========================

// Non-assignment form

(ExcInfo, bits(8*size)) MemA_with_priv_security(bits(32) address, integer size,
                                                AccType acctype, boolean privileged,
                                                boolean secure, boolean aligned)
    // Check alignment
    excInfo = DefaultExcInfo();
    if !IsAligned(address, size) then
        if HaveMainExt() then
            if secure then
                UFSR_S.UNALIGNED  = '1';
            else
                UFSR_NS.UNALIGNED = '1';
        // Create the exception. NOTE: If Main Extension is not implemented the fault
        // always escalates to a HardFault
        excInfo = CreateException(UsageFault, TRUE, secure);

    // Check permissions and get attributes
    if excInfo.fault == NoFault then
        (excInfo, memaddrdesc) = ValidateAddress(address, acctype, privileged, secure,
                                                 FALSE, aligned);

    if excInfo.fault == NoFault then
        // Memory array access, and sort out endianness
        (error, value) = _Mem(memaddrdesc, size);

        // Check if a synchronous BusFault occurred, async BusFaults are handled
```

```
        // in RaiseAsyncBusFault()
        if error then
            value = bits(8*size) UNKNOWN;
            if HaveMainExt() then
                case acctype of
                    when AccType_VECTABLE
                        HFSR.VECTTBL   = '1';
                        excInfo        = CreateException(HardFault, TRUE, AIRCR.BFHFNMINS ==
                            '0');
                    when AccType_STACK
                        BFSR.UNSTKERR  = '1';
                        excInfo        = CreateException(BusFault, FALSE, secure);
                    when AccType_NORMAL, AccType_MVE, AccType_ORDERED
                        BFAR.ADDRESS   = address;
                        BFSR.BFARVALID = '1';
                        BFSR.PRECISERR = '1';
                        // Generate BusFault exception if it cannot be ignored.
                        if !IsReqExcPriNeg(secure) || (CCR.BFHFNMIGN == '0') then
                            excInfo = CreateException(BusFault, FALSE, secure);
                    otherwise
                        // Some access types don't call this function
                        assert(FALSE);

        elsif BigEndian(address, size) then
            value = BigEndianReverse(value, size);

        // Check for Watch Point Match
        if IsDWTEnabled() then
            bits(32) dvalue = ZeroExtend(value);
            DWT_DataMatch(address, size, dvalue, TRUE, !secure);

    return (excInfo, value);

// Assignment form

ExcInfo MemA_with_priv_security(bits(32) address, integer size, AccType acctype,
                                boolean privileged, boolean secure, boolean aligned,
                                bits(8*size) value)
    // Check alignment
    excInfo = DefaultExcInfo();
    if !IsAligned(address, size) then
        if HaveMainExt() then
            if secure then
                UFSR_S.UNALIGNED  = '1';
            else
                UFSR_NS.UNALIGNED = '1';
        // Create the exception. NOTE: If Main Extension is not implemented the fault
        // always escalates to a HardFault
        excInfo = CreateException(UsageFault, TRUE, secure);

    // Check permissions and get attributes
    if excInfo.fault == NoFault then
        (excInfo, memaddrdesc) = ValidateAddress(address, acctype, privileged, secure,
                                                 TRUE, aligned);

    if excInfo.fault == NoFault then
        // Effect on exclusives
        if memaddrdesc.memattrs.shareable then
            ClearExclusiveByAddress(memaddrdesc.paddress,
                                    ProcessorID(), size);    // see Note

        // Check for Watch Point Match
        if IsDWTEnabled() then
            bits(32) dvalue = ZeroExtend(value);
            DWT_DataMatch(address, size, dvalue, FALSE, !secure);

        // Sort out endianness, then memory array access
        if BigEndian(address, size) then
            value = BigEndianReverse(value, size);
```

```
99
100         if _Mem(memaddrdesc, size, value) then
101             // Synchronous BusFault occurred. NOTE: async BusFaults are handled
102             // in RaiseAsyncBusFault()
103             if HaveMainExt() then
104                 case acctype of
105                     when AccType_STACK
106                         BFSR.STKERR    = '1';
107                         excInfo        = CreateException(BusFault, FALSE, secure);
108                     when AccType_LAZYFP
109                         BFSR.LSPERR    = '1';
110                         excInfo        = CreateException(BusFault, FALSE, secure);
111                     when AccType_NORMAL, AccType_MVE, AccType_ORDERED
112                         BFAR.ADDRESS   = address;
113                         BFSR.BFARVALID = '1';
114                         BFSR.PRECISERR = '1';
115                         // Generate BusFault exception if it cannot be ignored.
116                         if !IsReqExcPriNeg(secure) || (CCR.BFHFNMIGN == '0') then
117                             excInfo = CreateException(BusFault, FALSE, secure);
118                     otherwise
119                         // Some access types don't call this function
120                         assert(FALSE);
121     return excInfo;
```

## E2.1.265 MemD_with_priv_security

```
1  // MemD_with_priv_security()
2  // =========================
3
4  // Non-assignment form
5  (boolean, bits(8*size)) MemD_with_priv_security(AddressDescriptor attr, integer size)
6      // Debugger accesses always specify their required privilege/security levels, but can be
           demoted.
7      (secure, privileged, error) = DAPCheck(attr.paddress, attr.accattrs.ispriv, !attr.
           memattrs.NS, attr.accattrs.iswrite);
8
9      if !error then
10         (excInfo, memaddrdesc) = ValidateAddress(attr.paddress,
11                                                  AccType_DBG,
12                                                  privileged,
13                                                  secure,
14                                                  attr.accattrs.iswrite,
15                                                  IsAligned(attr.paddress, size));
16         // Inherit memory attributes from IMPDEF debugger interface if not
17         // accessed via UDE
18         secure = !attr.memattrs.NS && HaveSecurityExt();
19         if (secure && DHCSR.S_SUIDE  == '0') || (!secure && DHCSR.S_NSUIDE  == '0') then
20             memaddrdesc.memattrs         = attr.memattrs;
21             memaddrdesc.accattrs.acctype = attr.accattrs.acctype;
22         error = (excInfo.fault != NoFault);
23     if !error then
24         (error, value) = _Mem(memaddrdesc, size);
25     if error then
26         value = bits(8*size) UNKNOWN;
27
28     // No exception is thrown here since the debugger shouldn't be able to cause exceptions
           in the core.
29     // Instead, the caller should check against NoFault and return that information to the
           debugger.
30     return (error, value);
31
32
33 // Assignment form
34 boolean MemD_with_priv_security(AddressDescriptor attr, integer size, bits(8*size) value)
35     // Debugger accesses always specify their required privilege/security levels, but can be
           demoted.
36     (secure, privileged, error) = DAPCheck(attr.paddress, attr.accattrs.ispriv, !attr.
           memattrs.NS, attr.accattrs.iswrite);
```

```
37
38      if !error then
39          (excInfo, memaddrdesc) = ValidateAddress(attr.paddress,
40                                                   AccType_DBG,
41                                                   privileged,
42                                                   secure,
43                                                   attr.accattrs.iswrite,
44                                                   IsAligned(attr.paddress, size));
45          // Inherit memory attributes from IMPDEF debugger interface if not
46          // accessed via UDE
47          secure = !attr.memattrs.NS && HaveSecurityExt();
48          if (secure && DHCSR.S_SUIDE  == '0') || (!secure && DHCSR.S_NSUIDE  == '0') then
49              memaddrdesc.memattrs         = attr.memattrs;
50              memaddrdesc.accattrs.acctype = attr.accattrs.acctype;
51          error = (excInfo.fault != NoFault);
52      if !error then
53          error = _Mem(memaddrdesc, size, value);
54
55      // No exception is thrown here since the debugger shouldn't be able to cause exceptions
            in the core.
56      // Instead, the caller should check against NoFault and return that information to the
            debugger.
57      return error;
```

### E2.1.266 MemI

```
1   // MemI()
2   // ======
3
4   bits(16) MemI[bits(32) address]
5       // Check permissions and get attributes
6       // NOTE: The privilige flag passed to ValidateAddress may be overriden if
7       //       the security of the memory is different from the current security
8       //       state, eg when performing a Non-secure to Secure function call.
9       (excInfo, memaddrdesc) = ValidateAddress(address, AccType_IFETCH, FindPriv(),
10                                               IsSecure(), FALSE, TRUE);
11      if excInfo.fault == NoFault then
12          (error, value) = _Mem(memaddrdesc, 2);
13          if error then
14              value = bits(16) UNKNOWN;
15              BFSR.IBUSERR = '1';
16              // Create the exception. NOTE: If Main Extension is not implemented the fault
17              // always escalates to a HardFault
18              excInfo = CreateException(BusFault);
19      HandleException(excInfo);
20      if IsDWTEnabled() then DWT_InstructionMatch(address);
21      return value;
```

### E2.1.267 MemO

```
1   // MemO[] - non-assignment form
2   // ============================
3
4   bits(8*size) MemO[bits(32) address, integer size]
5       (excInfo, value) = MemA_with_priv_security(address, size, AccType_ORDERED,
6                                                  FindPriv(), IsSecure(), TRUE);
7       HandleException(excInfo);
8       return value;
9
10
11  // MemO[] - assignment form
12  // ========================
13
14  MemO[bits(32) address, integer size] = bits(8*size) value
15      excInfo = MemA_with_priv_security(address, size, AccType_ORDERED, FindPriv(),
16                                        IsSecure(), TRUE, value);
17      HandleException(excInfo);
```

### E2.1.268 MemoryAttributes

```
// v8-M Memory Attributes
type MemoryAttributes is (
    MemType memtype,
    DeviceType device,      // For Device memory
    bits(2) innerattrs,     // The possible encodings for each attributes field are as
        follows:
    bits(2) outerattrs,     // '00' = Non-cacheable; '01' = Write-Back
                            // '10' = Write-Through; '11' = RESERVED
    bits(2) innerhints,     // The possible encodings for the hints are as follows
    bits(2) outerhints,     // '00' = No-Allocate; '01' = Write-Allocate
                            // '10' = Read-Allocate; ;'11' = Read-Allocate and Write-Allocate
    boolean NS,             // TRUE if Non-secure, else FALSE
    boolean innertransient,
    boolean outertransient,
    boolean shareable,
    boolean outershareable
)
```

### E2.1.269 MemType

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device};
```

### E2.1.270 MemU

```
// MemU[]
// ======

// Non-assignment form, used for memory reads
// ==========================================

bits(8*size) MemU[bits(32) address, integer size]
    if HaveMainExt() then
        return MemU_with_priv[address, size, FindPriv()];
    else
        return MemA[address, size];


// Assignment form, used for memory writes
// =======================================

MemU[bits(32) address, integer size] = bits(8*size) value
    if HaveMainExt() then
        MemU_with_priv[address, size, FindPriv()] = value;
    else
        MemA[address, size] = value;
    return;
```

### E2.1.271 MemU_unpriv

```
// 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;
```

### E2.1.272 MemU_with_priv

```
// 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, TRUE];
    elsif CCR.UNALIGN_TRP == '1' then
        UFSR.UNALIGNED = '1';
        excInfo = CreateException(UsageFault);
        HandleException(excInfo);
    else // if unaligned access
        for i = 0 to size-1
            value<8*i+7:8*i> = MemA_with_priv[address+i, 1, privileged, FALSE];
        if BigEndian(address, size) 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, size, privileged, TRUE] = value;
    elsif CCR.UNALIGN_TRP == '1' then
        UFSR.UNALIGNED = '1';
        excInfo = CreateException(UsageFault);
        HandleException(excInfo);
    else // if unaligned access
        if BigEndian(address, size) then
            value = BigEndianReverse(value, size);
        for i = 0 to size-1
            MemA_with_priv[address+i, 1, privileged, FALSE] = value<8*i+7:8*i>;

    return;
```

### E2.1.273 MergeExcInfo

```
// MergeExcInfo()
// ==============

ExcInfo MergeExcInfo(ExcInfo a, ExcInfo b)
    // The ExcInfo structure is used to determine which exception should be
    // taken, and how it should be handled (mainly in the case of derived
    // exceptions).
    if    (b.fault == NoFault) || (a.isTerminal && !b.isTerminal) then
        exc = a;
    elsif (a.fault == NoFault) || (b.isTerminal && !a.isTerminal) then
        exc = b;
    elsif (a.fault == b.fault) && (a.isSecure == a.isSecure) then
        exc = a;
    else
        // Propagate the fault with the highest priority (lowest numerical
        // value).
        aPri = ExceptionPriority(a.fault, a.isSecure, FALSE);
        bPri = ExceptionPriority(b.fault, b.isSecure, FALSE);

        // Compare the exception priority values. Exception with the highest priority, which
```

```
        // is the lowest numerical value, is taken and the other exception may be pended.
        if aPri < bPri then
            exc  = a;
            pend = b;
        elsif bPri < aPri then
            exc  = b;
            pend = a;
        // If both priority values are equal, the exception numbers are compared.
        // The exception with the lowest exception number is taken and the other
        // exception may be pended.
        elsif a.fault < b.fault then
            exc  = a;
            pend = b;
        elsif b.fault < a.fault then
            exc  = b;
            pend = a;
        // If the two exception number are equal, the Secure exception is taken and the
        // Non-secure exception may be pended.
        elsif a.isSecure && !b.isSecure then
            exc  = a;
            pend = b;
        // In any other case exception (b) is taken and exception (a) is pended.
        else
            exc  = b;
            pend = a;

        // It is IMPLEMENTATION_DEFINED whether all exceptions generated are visible or not.
        // If visible, the highest priority exception will become active and lower priority
        // exceptions will get pended.
        if boolean IMPLEMENTATION_DEFINED "Overridden exceptions pended" then
            SetPending(pend.fault, pend.isSecure, TRUE);
    return exc;
```

## E2.1.274 min

```
// Min()
// =====

__overloaded integer Min(integer a, integer b)
    return if a <= b then a else b;

__overloaded real Min(real a, real b)
    return if a <= b then a else b;
```

## E2.1.275 MPUCheck

```
// MPUCheck()
// ==========

(MemoryAttributes, Permissions) MPUCheck(bits(32) address, AccType acctype,
    boolean ispriv, boolean secure)

    assert(HaveSecurityExt() || !secure);
    MemoryAttributes attributes;
    Permissions      perms;
    attributes = DefaultMemoryAttributes(address);
    perms      = DefaultPermissions(address);
    // assume no valid MPU region and not using default memory map
    hit         = FALSE;
    isPPBaccess = (address<31:20> == '111000000000');
    mpuCtrl     = if secure then MPU_CTRL_S else MPU_CTRL_NS;

    // Determine what MPU permissions should apply based on access type and MPU
    // configuration
    if acctype == AccType_VECTABLE || isPPBaccess then
        hit = TRUE;      // use default map for PPB and vector table lookups
    elsif acctype == AccType_DBG &&  secure && DHCSR.S_SUIDE  == '0' then
```

```
        hit = TRUE;      // use the debugger-provided memory attributes
    elsif acctype == AccType_DBG && !secure && DHCSR.S_NSUIDE == '0' then
        hit = TRUE;      // use the debugger-provided memory attributes
    elsif mpuCtrl.ENABLE == '0' then
        if mpuCtrl.HFNMIENA == '1' then UNPREDICTABLE;
        else hit = TRUE; // always use default map if MPU disabled
    elsif mpuCtrl.HFNMIENA == '0' && IsReqExcPriNeg(secure, acctype) then
        hit = TRUE;      // optionally use default for HardFault, NMI and FAULTMASK.
    else  // MPU is enabled so check each individual region
        if (mpuCtrl.PRIVDEFENA == '1') && ispriv then
            hit = TRUE;  // optional default as background for Privileged accesses

        regionMatched = FALSE;
        mpuType       = if secure then MPU_TYPE_S else MPU_TYPE_NS;
        for r = 0 to (UInt(mpuType.DREGION) - 1)
            if secure then
                rbar = __MPU_RBAR_S[r];
                rlar = __MPU_RLAR_S[r];
            else
                rbar = __MPU_RBAR_NS[r];
                rlar = __MPU_RLAR_NS[r];

            // MPU region enabled so perform checks
            if rlar.EN == '1' then
                if ((UInt(address) >= UInt(rbar.BASE  : '00000'))  &&
                    (UInt(address) <= UInt(rlar.LIMIT : '11111'))) then
                    // flag error if multiple regions match
                    if regionMatched then
                        perms.regionValid = FALSE;
                        perms.region      = Zeros(8);
                        hit               = FALSE;
                    else
                        regionMatched     = TRUE;
                        perms.ap          = rbar.AP;
                        if (rbar.XN == '1') || (ispriv && (rlar.PXN == '1')) then
                            perms.xn = '1';
                        else
                            perms.xn = '0';
                        perms.region      = r<7:0>;
                        perms.regionValid = TRUE;
                        hit               = TRUE;
                        sh                = rbar.SH;

                    // parsing MAIR0/1 Register fields
                    index         = UInt(rlar.AttrIndx);
                    mair          = (if secure then MPU_MAIR1_S  : MPU_MAIR0_S else
                                                    MPU_MAIR1_NS : MPU_MAIR0_NS);
                    attrfield     = mair<8*index+7:8*index>;
                    // decoding MAIR0/1 field and populating memory attributes
                    attributes    =  MAIRDecode(attrfield, sh);

    // MVE accesses to device memory are relaxed to GRE
    if acctype == AccType_MVE && attributes.memtype == MemType_Device then
        attributes.device = DeviceType_GRE;
    if address<31:29> == '111' then  // enforce System space execute never
        perms.xn = '1';
    if !hit then  // Access not allowed if no MPU match and use of default not enabled
        perms.apValid = FALSE;
    return (attributes, perms);
```

## E2.1.276 NextInstrAddr

```
// NextInstrAddr()
// ===============

bits(32) NextInstrAddr()
    return GetInstrExecState(1).FetchAddr;
```

### E2.1.277 NextInstrITState

```
// NextInstrITState()
// ==================

ITSTATEType NextInstrITState()
    if HaveMainExt() then
        nextState = GetInstrExecState(1).ITState;
    else
        nextState = Zeros(8);
    return nextState;
```

### E2.1.278 NoninvasiveDebugAllowed

```
// NoninvasiveDebugAllowed()
// =========================

boolean NoninvasiveDebugAllowed()
    return (ExternalNoninvasiveDebugEnabled() ||
            UnprivHaltingDebugAllowed(FALSE)  ||
            HaltingDebugAllowed());
```

### E2.1.279 ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

bits(N) Ones()
    return Ones(N);
```

### E2.1.280 PC

```
// PC - non-assignment form
// ========================
bits(32) PC
    return RName[RNamesPC];
```

### E2.1.281 PEMode

```
// The PE execution modes.

enumeration PEMode {PEMode_Thread, PEMode_Handler};
```

### E2.1.282 PendingDebugHalt

```
// PendingDebugHalt()
// ==================

boolean PendingDebugHalt()
    return CanHaltOnEvent(IsSecure()) && DHCSR.C_HALT == '1';
```

### E2.1.283 PendingDebugMonitor

```
// PendingDebugMonitor()
// =====================

boolean PendingDebugMonitor()
    // If the current execution priority is below DebugMonitor and generating a DebugMonitor
    // exception is allowed, and MON_PEND is set, then return TRUE. Otherwise return FALSE.
    return DEMCR.MON_PEND == '1' && CanPendMonitorOnEvent(IsSecure(), TRUE, FALSE);
```

### E2.1.284 PendingExceptionDetails

```
// PendingExceptionDetails
// =======================
// Determines whether to take a pending exception or not. This is done based
// on current execution priority and the priority of pending exceptions that
// are not masked by DHCSR.C_MASKINTS.
// Returns whether any pending exception is to be taken, and, if so, the
// exception number for the highest priority unmasked exception, and
// whether this exception is Secure.

(boolean, integer, boolean) PendingExceptionDetails();
```

### E2.1.285 PendReturnOperation

```
// PendReturnOperation()
// =====================

PendReturnOperation(bits(32) returnValue)
    _NextInstrAddr          = returnValue;
    _PCChanged              = TRUE;
    _PendingReturnOperation = TRUE;
    return;
```

### E2.1.286 Permissions

```
// Access permissions descriptor

type Permissions is (
    boolean apValid,     // TRUE when ap is valid, else FALSE
    bits(2) ap,          // Access Permission bits, if valid
    bit     xn,          // Execute Never bit
    boolean regionValid, // TRUE if the region number is valid, else FALSE
    bits(8) region       // The MPU region number, if valid
)
```

### E2.1.287 PMU_CounterIncrement

```
// PMU_CounterIncrement()
// ======================
// Increments PMU counters associated with the specified event as needed.

constant integer CYCLE_COUNTER_ID = 31;

PMU_CounterIncrement(PmuEventType eventId, integer counterId)
    // If the counter is disabled it does not need incrementing. Early-exit.
    if PMU_CTRL.E == '0' || DEMCR.TRCENA == '1' then
        return;

    // Frozen, counters do not increment
    if Sleeping || IsDebugState() || (!IsZero(PMU_OVSSET) && PMU_CTRL.FZO == '1') then
        return;

    pmuAllowedState = (!IsSecure() && NoninvasiveDebugAllowed()) ||
        SecureNoninvasiveDebugAllowed();

    // If the PMU_CTRL.DP bit is not set, the dedicated cycle counter is
    // enabled when in a prohibited state.
    if counterId == CYCLE_COUNTER_ID && PMU_CTRL.DP == '0' && eventId ==
        PmuEventType_CPU_CYCLES then
        pmuAllowedState = TRUE;

    // Prohibited, counters do not increment
    if !pmuAllowedState then
        return;
```

```
26
27      // The cycle counter will increment whenever it is enabled and there is a
28      // CPU_CYCLE event.
29      if counterId == CYCLE_COUNTER_ID then
30          if PMU_CNTENSET.C == '1' && eventId == PmuEventType_CPU_CYCLES then
31              newValue  = UInt(PMU_CCNTR) + 1;
32              PMU_CCNTR = newValue<31:0>;
33              - = PMU_HandleOverflow(counterId, newValue, 32);
34
35      // Other counters will increment if they are enabled, are configured to respond to that
            event
36      elsif PMU_CNTENSET.Pn<counterId> == '1' && PMU_EVTYPER[counterId] == PmuEvent(eventId)
            then
37          newValue = UInt(PMU_EVCNTR[counterId]) + 1;
38          PMU_EVCNTR[counterId].Counter = newValue<15:0>;
39          if PMU_HandleOverflow(counterId, newValue, 16) then
40              // If this is an EVEN counter, look at possible chaining
41              if counterId<0> == '0'                    &&
42                 PMU_CNTENSET.Pn<counterId + 1> == '1' &&
43                 PMU_EVTYPER[counterId + 1] == PmuEvent(PmuEventType_CHAIN) then
44                  // Configured as chaining counter, increment
45                  newValueChain = UInt(PMU_EVCNTR[counterId + 1]) + 1;
46                  PMU_EVCNTR[counterId + 1].Counter = newValueChain<15:0>;
47                  - = PMU_HandleOverflow(counterId + 1, newValueChain, 16);
48
49  PMU_CounterIncrement(PmuEventType eventId)
50      // If all counters are globally disable, they don't need incrementing. Early-exit.
51      if PMU_CTRL.E == '0' || DEMCR.TRCENA == '1' then
52          return;
53      for i = 0 to UInt(PMU_TYPE.N) - 1
54          PMU_CounterIncrement(eventId, i);
55      PMU_CounterIncrement(eventId, CYCLE_COUNTER_ID);
```

## E2.1.288 PMU_HandleOverflow

```
1   // PMU_HandleOverflow()
2   // ====================
3   // Handles the overflow of a specified counter.
4
5   boolean PMU_HandleOverflow(integer counterId, integer newValue, integer overflowBit)
6       // Handle trace-on-overflow if the lower 8-bits of any of the first 8 counters
7       // overflows. This only occurs if trace-on-overflow is enabled.
8       // If multiple trace packets are waiting to be issued, their contents can be
9       // merged into a single packet.
10      if counterId < 8 && PMU_CTRL.TRO == '1' && newValue<8> != (newValue - 1)<8> then
11          PMU_EmitTrace(counterId);
12
13      // Has the counter actually overflowed?
14      if newValue<overflowBit> == '1' then
15          PMU_OVSSET<counterId> = '1';
16          // If enabled, generate a debug event with 'PMU' syndrome
17          if PMU_INTENSET<counterId> == '1' then
18              isSecure = FALSE;
19              if DHCSR.C_PMOV == '1' && CanHaltOnEvent(isSecure) then
20                  DHCSR.C_HALT   = '1';
21                  DFSR.PMU       = '1';
22              elsif CanPendMonitorOnEvent(isSecure, FALSE, TRUE) then
23                  DEMCR.MON_PEND = '1';
24                  DFSR.PMU       = '1';
25          return TRUE;
26      return FALSE;
```

## E2.1.289 PmuEvent

```
1   // PmuEvent
2   // =========
3
```

```
// This PmuEvent function defines a mapping between a human-readable
// PMU Event and the corresponding integer event ID.

bits(32) PmuEvent(PmuEventType value)
    integer eventId;
    case value of
        when PmuEventType_SW_INCR                           eventId = 0x0;
        when PmuEventType_L1I_CACHE_REFILL                  eventId = 0x1;
        when PmuEventType_L1D_CACHE_REFILL                  eventId = 0x3;
        when PmuEventType_L1D_CACHE                         eventId = 0x4;
        when PmuEventType_LD_RETIRED                        eventId = 0x6;
        when PmuEventType_ST_RETIRED                        eventId = 0x7;
        when PmuEventType_INST_RETIRED                      eventId = 0x8;
        when PmuEventType_EXC_TAKEN                         eventId = 0x9;
        when PmuEventType_EXC_RETURN                        eventId = 0xa;
        when PmuEventType_PC_WRITE_RETIRED                  eventId = 0xc;
        when PmuEventType_BR_IMMED_RETIRED                  eventId = 0xd;
        when PmuEventType_BR_RETURN_RETIRED                 eventId = 0xe;
        when PmuEventType_UNALIGNED_LDST_RETIRED            eventId = 0xf;
        when PmuEventType_BR_MIS_PRED                       eventId = 0x10;
        when PmuEventType_CPU_CYCLES                        eventId = 0x11;
        when PmuEventType_BR_PRED                           eventId = 0x12;
        when PmuEventType_MEM_ACCESS                        eventId = 0x13;
        when PmuEventType_L1I_CACHE                         eventId = 0x14;
        when PmuEventType_L1D_CACHE_WB                      eventId = 0x15;
        when PmuEventType_L2D_CACHE                         eventId = 0x16;
        when PmuEventType_L2D_CACHE_REFILL                  eventId = 0x17;
        when PmuEventType_L2D_CACHE_WB                      eventId = 0x18;
        when PmuEventType_BUS_ACCESS                        eventId = 0x19;
        when PmuEventType_MEMORY_ERROR                      eventId = 0x1a;
        when PmuEventType_INST_SPEC                         eventId = 0x1b;
        when PmuEventType_BUS_CYCLES                        eventId = 0x1d;
        when PmuEventType_CHAIN                             eventId = 0x1e;
        when PmuEventType_L1D_CACHE_ALLOCATE                eventId = 0x1f;
        when PmuEventType_L2D_CACHE_ALLOCATE                eventId = 0x20;
        when PmuEventType_BR_RETIRED                        eventId = 0x21;
        when PmuEventType_BR_MIS_PRED_RETIRED               eventId = 0x22;
        when PmuEventType_STALL_FRONTEND                    eventId = 0x23;
        when PmuEventType_STALL_BACKEND                     eventId = 0x24;
        when PmuEventType_L2I_CACHE                         eventId = 0x27;
        when PmuEventType_L2I_CACHE_REFILL                  eventId = 0x28;
        when PmuEventType_L3D_CACHE_ALLOCATE                eventId = 0x29;
        when PmuEventType_L3D_CACHE_REFILL                  eventId = 0x2a;
        when PmuEventType_L3D_CACHE                         eventId = 0x2b;
        when PmuEventType_L3D_CACHE_WB                      eventId = 0x2c;
        when PmuEventType_LL_CACHE_RD                       eventId = 0x36;
        when PmuEventType_LL_CACHE_MISS_RD                  eventId = 0x37;
        when PmuEventType_L1D_CACHE_MISS_RD                 eventId = 0x39;
        when PmuEventType_OP_COMPLETE                       eventId = 0x3a;
        when PmuEventType_OP_SPEC                           eventId = 0x3b;
        when PmuEventType_STALL                             eventId = 0x3c;
        when PmuEventType_STALL_OP_BACKEND                  eventId = 0x3d;
        when PmuEventType_STALL_OP_FRONTEND                 eventId = 0x3e;
        when PmuEventType_STALL_OP                          eventId = 0x3f;
        when PmuEventType_L1D_CACHE_RD                      eventId = 0x40;
        when PmuEventType_LE_RETIRED                        eventId = 0x100;
        when PmuEventType_LE_SPEC                           eventId = 0x101;
        when PmuEventType_BF_RETIRED                        eventId = 0x104;
        when PmuEventType_BF_SPEC                           eventId = 0x105;
        when PmuEventType_LE_CANCEL                         eventId = 0x108;
        when PmuEventType_BF_CANCEL                         eventId = 0x109;
        when PmuEventType_SE_CALL_S                         eventId = 0x114;
        when PmuEventType_SE_CALL_NS                        eventId = 0x115;
        when PmuEventType_MVE_INST_RETIRED                  eventId = 0x200;
        when PmuEventType_MVE_INST_SPEC                     eventId = 0x201;
        when PmuEventType_MVE_FP_RETIRED                    eventId = 0x204;
        when PmuEventType_MVE_FP_SPEC                       eventId = 0x205;
        when PmuEventType_MVE_FP_HP_RETIRED                 eventId = 0x208;
        when PmuEventType_MVE_FP_HP_SPEC                    eventId = 0x209;
```

```
        when PmuEventType_MVE_FP_SP_RETIRED                     eventId = 0x20c;
        when PmuEventType_MVE_FP_SP_SPEC                        eventId = 0x20d;
        when PmuEventType_MVE_FP_MAC_RETIRED                    eventId = 0x214;
        when PmuEventType_MVE_FP_MAC_SPEC                       eventId = 0x215;
        when PmuEventType_MVE_INT_RETIRED                       eventId = 0x224;
        when PmuEventType_MVE_INT_SPEC                          eventId = 0x225;
        when PmuEventType_MVE_INT_MAC_RETIRED                   eventId = 0x228;
        when PmuEventType_MVE_INT_MAC_SPEC                      eventId = 0x229;
        when PmuEventType_MVE_LDST_RETIRED                      eventId = 0x238;
        when PmuEventType_MVE_LDST_SPEC                         eventId = 0x239;
        when PmuEventType_MVE_LD_RETIRED                        eventId = 0x23c;
        when PmuEventType_MVE_LD_SPEC                           eventId = 0x23d;
        when PmuEventType_MVE_ST_RETIRED                        eventId = 0x240;
        when PmuEventType_MVE_ST_SPEC                           eventId = 0x241;
        when PmuEventType_MVE_LDST_CONTIG_RETIRED               eventId = 0x244;
        when PmuEventType_MVE_LDST_CONTIG_SPEC                  eventId = 0x245;
        when PmuEventType_MVE_LD_CONTIG_RETIRED                 eventId = 0x248;
        when PmuEventType_MVE_LD_CONTIG_SPEC                    eventId = 0x249;
        when PmuEventType_MVE_ST_CONTIG_RETIRED                 eventId = 0x24c;
        when PmuEventType_MVE_ST_CONTIG_SPEC                    eventId = 0x24d;
        when PmuEventType_MVE_LDST_NONCONTIG_RETIRED            eventId = 0x250;
        when PmuEventType_MVE_LDST_NONCONTIG_SPEC               eventId = 0x251;
        when PmuEventType_MVE_LD_NONCONTIG_RETIRED              eventId = 0x254;
        when PmuEventType_MVE_LD_NONCONTIG_SPEC                 eventId = 0x255;
        when PmuEventType_MVE_ST_NONCONTIG_RETIRED              eventId = 0x258;
        when PmuEventType_MVE_ST_NONCONTIG_SPEC                 eventId = 0x259;
        when PmuEventType_MVE_LDST_MULTI_RETIRED                eventId = 0x25c;
        when PmuEventType_MVE_LDST_MULTI_SPEC                   eventId = 0x25d;
        when PmuEventType_MVE_LD_MULTI_RETIRED                  eventId = 0x260;
        when PmuEventType_MVE_LD_MULTI_SPEC                     eventId = 0x261;
        when PmuEventType_MVE_ST_MULTI_RETIRED                  eventId = 0x264;
        when PmuEventType_MVE_ST_MULTI_SPEC                     eventId = 0x265;
        when PmuEventType_MVE_LDST_UNALIGNED_RETIRED            eventId = 0x28c;
        when PmuEventType_MVE_LDST_UNALIGNED_SPEC               eventId = 0x28d;
        when PmuEventType_MVE_LD_UNALIGNED_RETIRED              eventId = 0x290;
        when PmuEventType_MVE_LD_UNALIGNED_SPEC                 eventId = 0x291;
        when PmuEventType_MVE_ST_UNALIGNED_RETIRED              eventId = 0x294;
        when PmuEventType_MVE_ST_UNALIGNED_SPEC                 eventId = 0x295;
        when PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_RETIRED  eventId = 0x298;
        when PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_SPEC     eventId = 0x299;
        when PmuEventType_MVE_VREDUCE_RETIRED                   eventId = 0x2a0;
        when PmuEventType_MVE_VREDUCE_SPEC                      eventId = 0x2a1;
        when PmuEventType_MVE_VREDUCE_FP_RETIRED                eventId = 0x2a4;
        when PmuEventType_MVE_VREDUCE_FP_SPEC                   eventId = 0x2a5;
        when PmuEventType_MVE_VREDUCE_INT_RETIRED               eventId = 0x2a8;
        when PmuEventType_MVE_VREDUCE_INT_SPEC                  eventId = 0x2a9;
        when PmuEventType_MVE_PRED                              eventId = 0x2b8;
        when PmuEventType_MVE_STALL                             eventId = 0x2cc;
        when PmuEventType_MVE_STALL_RESOURCE                    eventId = 0x2cd;
        when PmuEventType_MVE_STALL_RESOURCE_MEM                eventId = 0x2ce;
        when PmuEventType_MVE_STALL_RESOURCE_FP                 eventId = 0x2cf;
        when PmuEventType_MVE_STALL_RESOURCE_INT                eventId = 0x2d0;
        when PmuEventType_MVE_STALL_BREAK                       eventId = 0x2d3;
        when PmuEventType_MVE_STALL_DEPENDENCY                  eventId = 0x2d4;
        when PmuEventType_ITCM_ACCESS                           eventId = 0x4007;
        when PmuEventType_DTCM_ACCESS                           eventId = 0x4008;
        // Events above 0xFFFF are reserved for
        // IMPLEMENTATION DEFINED events
        otherwise
            eventId = -1;
    return eventId<31:0>;

PmuEventType PmuEvent(bits(32) value)
    PmuEventType eventId;
    case UInt(value) of
        when 0x0   eventId  = PmuEventType_SW_INCR;
        when 0x1   eventId  = PmuEventType_L1I_CACHE_REFILL;
        when 0x3   eventId  = PmuEventType_L1D_CACHE_REFILL;
        when 0x4   eventId  = PmuEventType_L1D_CACHE;
```

```
        when 0x6   eventId  = PmuEventType_LD_RETIRED;
        when 0x7   eventId  = PmuEventType_ST_RETIRED;
        when 0x8   eventId  = PmuEventType_INST_RETIRED;
        when 0x9   eventId  = PmuEventType_EXC_TAKEN;
        when 0xa   eventId  = PmuEventType_EXC_RETURN;
        when 0xc   eventId  = PmuEventType_PC_WRITE_RETIRED;
        when 0xd   eventId  = PmuEventType_BR_IMMED_RETIRED;
        when 0xe   eventId  = PmuEventType_BR_RETURN_RETIRED;
        when 0xf   eventId  = PmuEventType_UNALIGNED_LDST_RETIRED;
        when 0x10  eventId  = PmuEventType_BR_MIS_PRED;
        when 0x11  eventId  = PmuEventType_CPU_CYCLES;
        when 0x12  eventId  = PmuEventType_BR_PRED;
        when 0x13  eventId  = PmuEventType_MEM_ACCESS;
        when 0x14  eventId  = PmuEventType_L1I_CACHE;
        when 0x15  eventId  = PmuEventType_L1D_CACHE_WB;
        when 0x16  eventId  = PmuEventType_L2D_CACHE;
        when 0x17  eventId  = PmuEventType_L2D_CACHE_REFILL;
        when 0x18  eventId  = PmuEventType_L2D_CACHE_WB;
        when 0x19  eventId  = PmuEventType_BUS_ACCESS;
        when 0x1a  eventId  = PmuEventType_MEMORY_ERROR;
        when 0x1b  eventId  = PmuEventType_INST_SPEC;
        when 0x1d  eventId  = PmuEventType_BUS_CYCLES;
        when 0x1e  eventId  = PmuEventType_CHAIN;
        when 0x1f  eventId  = PmuEventType_L1D_CACHE_ALLOCATE;
        when 0x20  eventId  = PmuEventType_L2D_CACHE_ALLOCATE;
        when 0x21  eventId  = PmuEventType_BR_RETIRED;
        when 0x22  eventId  = PmuEventType_BR_MIS_PRED_RETIRED;
        when 0x23  eventId  = PmuEventType_STALL_FRONTEND;
        when 0x24  eventId  = PmuEventType_STALL_BACKEND;
        when 0x27  eventId  = PmuEventType_L2I_CACHE;
        when 0x28  eventId  = PmuEventType_L2I_CACHE_REFILL;
        when 0x29  eventId  = PmuEventType_L3D_CACHE_ALLOCATE;
        when 0x2a  eventId  = PmuEventType_L3D_CACHE_REFILL;
        when 0x2b  eventId  = PmuEventType_L3D_CACHE;
        when 0x2c  eventId  = PmuEventType_L3D_CACHE_WB;
        when 0x36  eventId  = PmuEventType_LL_CACHE_RD;
        when 0x37  eventId  = PmuEventType_LL_CACHE_MISS_RD;
        when 0x39  eventId  = PmuEventType_L1D_CACHE_MISS_RD;
        when 0x3a  eventId  = PmuEventType_OP_COMPLETE;
        when 0x3b  eventId  = PmuEventType_OP_SPEC;
        when 0x3c  eventId  = PmuEventType_STALL;
        when 0x3d  eventId  = PmuEventType_STALL_OP_BACKEND;
        when 0x3e  eventId  = PmuEventType_STALL_OP_FRONTEND;
        when 0x3f  eventId  = PmuEventType_STALL_OP;
        when 0x40  eventId  = PmuEventType_L1D_CACHE_RD;
        when 0x100 eventId  = PmuEventType_LE_RETIRED;
        when 0x101 eventId  = PmuEventType_LE_SPEC;
        when 0x104 eventId  = PmuEventType_BF_RETIRED;
        when 0x105 eventId  = PmuEventType_BF_SPEC;
        when 0x108 eventId  = PmuEventType_LE_CANCEL;
        when 0x109 eventId  = PmuEventType_BF_CANCEL;
        when 0x114 eventId  = PmuEventType_SE_CALL_S;
        when 0x115 eventId  = PmuEventType_SE_CALL_NS;
        when 0x200 eventId  = PmuEventType_MVE_INST_RETIRED;
        when 0x201 eventId  = PmuEventType_MVE_INST_SPEC;
        when 0x204 eventId  = PmuEventType_MVE_FP_RETIRED;
        when 0x205 eventId  = PmuEventType_MVE_FP_SPEC;
        when 0x208 eventId  = PmuEventType_MVE_FP_HP_RETIRED;
        when 0x209 eventId  = PmuEventType_MVE_FP_HP_SPEC;
        when 0x20c eventId  = PmuEventType_MVE_FP_SP_RETIRED;
        when 0x20d eventId  = PmuEventType_MVE_FP_SP_SPEC;
        when 0x214 eventId  = PmuEventType_MVE_FP_MAC_RETIRED;
        when 0x215 eventId  = PmuEventType_MVE_FP_MAC_SPEC;
        when 0x224 eventId  = PmuEventType_MVE_INT_RETIRED;
        when 0x225 eventId  = PmuEventType_MVE_INT_SPEC;
        when 0x228 eventId  = PmuEventType_MVE_INT_MAC_RETIRED;
        when 0x229 eventId  = PmuEventType_MVE_INT_MAC_SPEC;
        when 0x238 eventId  = PmuEventType_MVE_LDST_RETIRED;
        when 0x239 eventId  = PmuEventType_MVE_LDST_SPEC;
```

```
        when 0x23c eventId  = PmuEventType_MVE_LD_RETIRED;
        when 0x23d eventId  = PmuEventType_MVE_LD_SPEC;
        when 0x240 eventId  = PmuEventType_MVE_ST_RETIRED;
        when 0x241 eventId  = PmuEventType_MVE_ST_SPEC;
        when 0x244 eventId  = PmuEventType_MVE_LDST_CONTIG_RETIRED;
        when 0x245 eventId  = PmuEventType_MVE_LDST_CONTIG_SPEC;
        when 0x248 eventId  = PmuEventType_MVE_LD_CONTIG_RETIRED;
        when 0x249 eventId  = PmuEventType_MVE_LD_CONTIG_SPEC;
        when 0x24c eventId  = PmuEventType_MVE_ST_CONTIG_RETIRED;
        when 0x24d eventId  = PmuEventType_MVE_ST_CONTIG_SPEC;
        when 0x250 eventId  = PmuEventType_MVE_LDST_NONCONTIG_RETIRED;
        when 0x251 eventId  = PmuEventType_MVE_LDST_NONCONTIG_SPEC;
        when 0x254 eventId  = PmuEventType_MVE_LD_NONCONTIG_RETIRED;
        when 0x255 eventId  = PmuEventType_MVE_LD_NONCONTIG_SPEC;
        when 0x258 eventId  = PmuEventType_MVE_ST_NONCONTIG_RETIRED;
        when 0x259 eventId  = PmuEventType_MVE_ST_NONCONTIG_SPEC;
        when 0x25c eventId  = PmuEventType_MVE_LDST_MULTI_RETIRED;
        when 0x25d eventId  = PmuEventType_MVE_LDST_MULTI_SPEC;
        when 0x260 eventId  = PmuEventType_MVE_LD_MULTI_RETIRED;
        when 0x261 eventId  = PmuEventType_MVE_LD_MULTI_SPEC;
        when 0x264 eventId  = PmuEventType_MVE_ST_MULTI_RETIRED;
        when 0x265 eventId  = PmuEventType_MVE_ST_MULTI_SPEC;
        when 0x28c eventId  = PmuEventType_MVE_LDST_UNALIGNED_RETIRED;
        when 0x28d eventId  = PmuEventType_MVE_LDST_UNALIGNED_SPEC;
        when 0x290 eventId  = PmuEventType_MVE_LD_UNALIGNED_RETIRED;
        when 0x291 eventId  = PmuEventType_MVE_LD_UNALIGNED_SPEC;
        when 0x294 eventId  = PmuEventType_MVE_ST_UNALIGNED_RETIRED;
        when 0x295 eventId  = PmuEventType_MVE_ST_UNALIGNED_SPEC;
        when 0x298 eventId  = PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_RETIRED;
        when 0x299 eventId  = PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_SPEC;
        when 0x2a0 eventId  = PmuEventType_MVE_VREDUCE_RETIRED;
        when 0x2a1 eventId  = PmuEventType_MVE_VREDUCE_SPEC;
        when 0x2a4 eventId  = PmuEventType_MVE_VREDUCE_FP_RETIRED;
        when 0x2a5 eventId  = PmuEventType_MVE_VREDUCE_FP_SPEC;
        when 0x2a8 eventId  = PmuEventType_MVE_VREDUCE_INT_RETIRED;
        when 0x2a9 eventId  = PmuEventType_MVE_VREDUCE_INT_SPEC;
        when 0x2b8 eventId  = PmuEventType_MVE_PRED;
        when 0x2cc eventId  = PmuEventType_MVE_STALL;
        when 0x2cd eventId  = PmuEventType_MVE_STALL_RESOURCE;
        when 0x2ce eventId  = PmuEventType_MVE_STALL_RESOURCE_MEM;
        when 0x2cf eventId  = PmuEventType_MVE_STALL_RESOURCE_FP;
        when 0x2d0 eventId  = PmuEventType_MVE_STALL_RESOURCE_INT;
        when 0x2d3 eventId  = PmuEventType_MVE_STALL_BREAK;
        when 0x2d4 eventId  = PmuEventType_MVE_STALL_DEPENDENCY;
        when 0x4007 eventId = PmuEventType_ITCM_ACCESS;
        when 0x4008 eventId = PmuEventType_DTCM_ACCESS;
        // Events above 0xFFFF are reserved for
        // IMPLEMENTATION DEFINED events
        otherwise
            eventId = PmuEventType_NONE;
    return eventId;
```

## E2.1.290 PmuEventType

```
// Enumeration of the supported PMU Events
enumeration PmuEventType {
    PmuEventType_NONE,
    PmuEventType_SW_INCR,
    PmuEventType_L1I_CACHE_REFILL,
    PmuEventType_L1D_CACHE_REFILL,
    PmuEventType_L1D_CACHE,
    PmuEventType_LD_RETIRED,
    PmuEventType_ST_RETIRED,
    PmuEventType_INST_RETIRED,
    PmuEventType_EXC_TAKEN,
    PmuEventType_EXC_RETURN,
    PmuEventType_PC_WRITE_RETIRED,
    PmuEventType_BR_IMMED_RETIRED,
```

```
    PmuEventType_BR_RETURN_RETIRED,
    PmuEventType_UNALIGNED_LDST_RETIRED,
    PmuEventType_BR_MIS_PRED,
    PmuEventType_CPU_CYCLES,
    PmuEventType_BR_PRED,
    PmuEventType_MEM_ACCESS,
    PmuEventType_L1I_CACHE,
    PmuEventType_L1D_CACHE_WB,
    PmuEventType_L2D_CACHE,
    PmuEventType_L2D_CACHE_REFILL,
    PmuEventType_L2D_CACHE_WB,
    PmuEventType_BUS_ACCESS,
    PmuEventType_MEMORY_ERROR,
    PmuEventType_INST_SPEC,
    PmuEventType_BUS_CYCLES,
    PmuEventType_CHAIN,
    PmuEventType_L1D_CACHE_ALLOCATE,
    PmuEventType_L2D_CACHE_ALLOCATE,
    PmuEventType_BR_RETIRED,
    PmuEventType_BR_MIS_PRED_RETIRED,
    PmuEventType_STALL_FRONTEND,
    PmuEventType_STALL_BACKEND,
    PmuEventType_L2I_CACHE,
    PmuEventType_L2I_CACHE_REFILL,
    PmuEventType_L3D_CACHE_ALLOCATE,
    PmuEventType_L3D_CACHE_REFILL,
    PmuEventType_L3D_CACHE,
    PmuEventType_L3D_CACHE_WB,
    PmuEventType_LL_CACHE_RD,
    PmuEventType_LL_CACHE_MISS_RD,
    PmuEventType_L1D_CACHE_MISS_RD,
    PmuEventType_OP_COMPLETE,
    PmuEventType_OP_SPEC,
    PmuEventType_STALL,
    PmuEventType_STALL_OP_BACKEND,
    PmuEventType_STALL_OP_FRONTEND,
    PmuEventType_STALL_OP,
    PmuEventType_L1D_CACHE_RD,
    PmuEventType_LE_RETIRED,
    PmuEventType_LE_SPEC,
    PmuEventType_BF_RETIRED,
    PmuEventType_BF_SPEC,
    PmuEventType_LE_CANCEL,
    PmuEventType_BF_CANCEL,
    PmuEventType_SE_CALL_S,
    PmuEventType_SE_CALL_NS,
    PmuEventType_MVE_INST_RETIRED,
    PmuEventType_MVE_INST_SPEC,
    PmuEventType_MVE_FP_RETIRED,
    PmuEventType_MVE_FP_SPEC,
    PmuEventType_MVE_FP_HP_RETIRED,
    PmuEventType_MVE_FP_HP_SPEC,
    PmuEventType_MVE_FP_SP_RETIRED,
    PmuEventType_MVE_FP_SP_SPEC,
    PmuEventType_MVE_FP_MAC_RETIRED,
    PmuEventType_MVE_FP_MAC_SPEC,
    PmuEventType_MVE_INT_RETIRED,
    PmuEventType_MVE_INT_SPEC,
    PmuEventType_MVE_INT_MAC_RETIRED,
    PmuEventType_MVE_INT_MAC_SPEC,
    PmuEventType_MVE_LDST_RETIRED,
    PmuEventType_MVE_LDST_SPEC,
    PmuEventType_MVE_LD_RETIRED,
    PmuEventType_MVE_LD_SPEC,
    PmuEventType_MVE_ST_RETIRED,
    PmuEventType_MVE_ST_SPEC,
    PmuEventType_MVE_LDST_CONTIG_RETIRED,
    PmuEventType_MVE_LDST_CONTIG_SPEC,
    PmuEventType_MVE_LD_CONTIG_RETIRED,
```

```
    PmuEventType_MVE_LD_CONTIG_SPEC,
    PmuEventType_MVE_ST_CONTIG_RETIRED,
    PmuEventType_MVE_ST_CONTIG_SPEC,
    PmuEventType_MVE_LDST_NONCONTIG_RETIRED,
    PmuEventType_MVE_LDST_NONCONTIG_SPEC,
    PmuEventType_MVE_LD_NONCONTIG_RETIRED,
    PmuEventType_MVE_LD_NONCONTIG_SPEC,
    PmuEventType_MVE_ST_NONCONTIG_RETIRED,
    PmuEventType_MVE_ST_NONCONTIG_SPEC,
    PmuEventType_MVE_LDST_MULTI_RETIRED,
    PmuEventType_MVE_LDST_MULTI_SPEC,
    PmuEventType_MVE_LD_MULTI_RETIRED,
    PmuEventType_MVE_LD_MULTI_SPEC,
    PmuEventType_MVE_ST_MULTI_RETIRED,
    PmuEventType_MVE_ST_MULTI_SPEC,
    PmuEventType_MVE_LDST_UNALIGNED_RETIRED,
    PmuEventType_MVE_LDST_UNALIGNED_SPEC,
    PmuEventType_MVE_LD_UNALIGNED_RETIRED,
    PmuEventType_MVE_LD_UNALIGNED_SPEC,
    PmuEventType_MVE_ST_UNALIGNED_RETIRED,
    PmuEventType_MVE_ST_UNALIGNED_SPEC,
    PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_RETIRED,
    PmuEventType_MVE_LDST_UNALIGNED_NONCONTIG_SPEC,
    PmuEventType_MVE_VREDUCE_RETIRED,
    PmuEventType_MVE_VREDUCE_SPEC,
    PmuEventType_MVE_VREDUCE_FP_RETIRED,
    PmuEventType_MVE_VREDUCE_FP_SPEC,
    PmuEventType_MVE_VREDUCE_INT_RETIRED,
    PmuEventType_MVE_VREDUCE_INT_SPEC,
    PmuEventType_MVE_PRED,
    PmuEventType_MVE_STALL,
    PmuEventType_MVE_STALL_RESOURCE,
    PmuEventType_MVE_STALL_RESOURCE_MEM,
    PmuEventType_MVE_STALL_RESOURCE_FP,
    PmuEventType_MVE_STALL_RESOURCE_INT,
    PmuEventType_MVE_STALL_BREAK,
    PmuEventType_MVE_STALL_DEPENDENCY,
    PmuEventType_ITCM_ACCESS,
    PmuEventType_DTCM_ACCESS
    // Other implementation-specific events may be defined here
};
```

### E2.1.291 PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = Zeros(M) : op2;
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

### E2.1.292 PopStack

```
// PopStack()
// ==========

ExcInfo PopStack(EXC_RETURN_Type excReturn)
    constant integer intCallerFrameSize = 0x20;
    constant integer intCalleeFrameSize = 0x28;
    constant integer fpCallerFrameSize  = 0x48;

    // NOTE: All stack accesses are performed as Unprivileged accesses if
    // returning to thread mode and CONTROL.nPRIV is 1 for the destination
```

```
    // Security state.
    mode     = if excReturn.Mode == '1' then PEMode_Thread else PEMode_Handler;
    toSecure = HaveSecurityExt() && excReturn.S == '1';
    spName   = LookUpSP_with_security_mode(toSecure, mode);
    frameptr = _SP(spName);
    if !IsAligned(frameptr, 8) then UNPREDICTABLE;

    // only stack locations, not the load order, are architected

    // Pop the callee saved registers, when returning from a Non-secure exception
    // or a Secure one that followed a Non-secure one and therefore still has
    // the callee register state on the stack.
    exc = DefaultExcInfo();
    if toSecure && (excReturn.ES == '0' ||
                    excReturn.DCRS == '0') then
        // Check the integrity signature, and if so is it correct
        expectedSig = 0xFEFA125B<31:0>;
        if HaveMveOrFPExt() then
            expectedSig<0> = excReturn.FType;
        (exc, integritySig) = Stack(frameptr, 0x0, spName, mode);
        if exc.fault == NoFault && integritySig != expectedSig then
            if HaveMainExt() then
                SFSR.INVIS = '1';
            // Create the exception. NOTE: If Main Extension is not implemented the fault
            // always escalates to a HardFault
            return CreateException(SecureFault);

        if exc.fault == NoFault then (exc, R[4] ) = Stack(frameptr, 0x8,  spName, mode);
        if exc.fault == NoFault then (exc, R[5] ) = Stack(frameptr, 0xC,  spName, mode);
        if exc.fault == NoFault then (exc, R[6] ) = Stack(frameptr, 0x10, spName, mode);
        if exc.fault == NoFault then (exc, R[7] ) = Stack(frameptr, 0x14, spName, mode);
        if exc.fault == NoFault then (exc, R[8] ) = Stack(frameptr, 0x18, spName, mode);
        if exc.fault == NoFault then (exc, R[9] ) = Stack(frameptr, 0x1C, spName, mode);
        if exc.fault == NoFault then (exc, R[10]) = Stack(frameptr, 0x20, spName, mode);
        if exc.fault == NoFault then (exc, R[11]) = Stack(frameptr, 0x24, spName, mode);
        frameptr = frameptr + intCalleeFrameSize;

    // Unstack the caller saved regs, possibly including the FP regs
    RETPSR_Type psr;
    if exc.fault == NoFault then (exc, R[0] ) = Stack(frameptr, 0x0,  spName, mode);
    if exc.fault == NoFault then (exc, R[1] ) = Stack(frameptr, 0x4,  spName, mode);
    if exc.fault == NoFault then (exc, R[2] ) = Stack(frameptr, 0x8,  spName, mode);
    if exc.fault == NoFault then (exc, R[3] ) = Stack(frameptr, 0xC,  spName, mode);
    if exc.fault == NoFault then (exc, R[12]) = Stack(frameptr, 0x10, spName, mode);
    if exc.fault == NoFault then (exc, LR   ) = Stack(frameptr, 0x14, spName, mode);
    if exc.fault == NoFault then (exc, pc   ) = Stack(frameptr, 0x18, spName, mode);
    if exc.fault == NoFault then (exc, psr  ) = Stack(frameptr, 0x1C, spName, mode);
    frameOffset = intCallerFrameSize;
    BranchTo(pc, TRUE);

    // Check the XPSR value that has been unstacked is consistent with the mode
    // being returned to
    excNum = UInt(psr.Exception);
    if (exc.fault == NoFault) &&
       ((mode == PEMode_Handler) == (excNum == 0)) then
        if HaveMainExt() then
            UFSR.INVPC = '1';
        // Create the exception. NOTE: If Main Extension is not implemented the fault
        // always escalates to a HardFault
        return CreateException(UsageFault);

    // The IPSR value is set as UNKNOWN if the unstacked IPSR value is not supported by the
        PE
    validIPSR = excNum IN {0, 1, NMI, HardFault, SVCall, PendSV, SysTick};
    if !validIPSR && HaveMainExt() then
        validIPSR = excNum IN {MemManage, BusFault, UsageFault, SecureFault, DebugMonitor};

    // Check also whether excNum is an external interupt supported by PE
    if !validIPSR && !IsIrqValid(excNum) then
```

```
        psr.Exception = bits(9) UNKNOWN;

    if HaveMveOrFPExt() then
        if excReturn.FType == '0' then
            // Raise a fault and skip Floating-point operations if requested to expose
            // Secure Floating-point state to the Non-secure code.
            if !toSecure && FPCCR_S.LSPACT == '1' then
                SFSR.LSERR = '1';
                newExc     = CreateException(SecureFault);
                // It is IMPLEMENTATION DEFINED whether a MemFault is dropped if
                // a SecureFault is generated subsequently. If the MemFault is
                // not dropped the exceptions will be taken based on exception
                // priority as described in MergeExcInfo()
                if boolean IMPLEMENTATION_DEFINED "Drop previously generated exceptions" then
                    exc = newExc;
                else
                    exc = MergeExcInfo(exc, newExc);
            else
                lspact = if toSecure then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
                if lspact == '1' then // state in FP is still valid
                    if exc.fault == NoFault then
                        if toSecure then
                            FPCCR_S.LSPACT  = '0';
                        else
                            FPCCR_NS.LSPACT = '0';
                else
                    if exc.fault == NoFault then
                        nPriv  = if toSecure then CONTROL_S.nPRIV else CONTROL_NS.nPRIV;
                        isPriv = mode == PEMode_Handler || nPriv == '0';
                        exc    = CheckCPEnabled(10, isPriv, toSecure);

                    // If an implementation abandons the unstacking of the Floating-point
                    // Extension registers and to tail chain into a fault or late arriving
                    // interrupt it must clear any Floating-point registers that
                    // would have been unstacked.
                    // NOTE: The requirment to clear the registers only applies
                    // to implementations that include the Security Extensions.
                    // The Floating-point Extension registers that would have been unstacked
                        become
                    // UNKNOWN in implementations that do not include the
                    // Security Extensions.
                    if exc.fault == NoFault then
                        for i = 0 to 15
                            if exc.fault == NoFault then
                                (exc, S[i]) = Stack(frameptr, frameOffset + (4*i), spName,
                                    mode);
                        if exc.fault == NoFault then
                            (exc, FPSCR) = Stack(frameptr, frameOffset + 0x40, spName, mode);
                        if HaveMve() && exc.fault == NoFault then
                            (exc, VPR)   = Stack(frameptr, frameOffset + 0x44, spName, mode);
                        frameOffset = frameOffset + fpCallerFrameSize;

                        pushFPCalleeRegs = toSecure && FPCCR_S.TS == '1';
                        if pushFPCalleeRegs then
                            for i = 0 to 15
                                if exc.fault == NoFault then
                                    (exc, S[i+16]) = Stack(frameptr, frameOffset + (4*i),
                                        spName, mode);

                        if exc.fault != NoFault then
                            InvalidateFPRegs(HaveSecurityExt(), pushFPCalleeRegs);

        CONTROL.FPCA = NOT(excReturn.FType);

    // If there was not a fault then move the stack pointer to consume the
    // exception stack frame. NOTE: If a exception return fault occurs and
    // results in a lockup the stack pointer is updated. This special case is
    // handled at the point lockup is entered and not here.
    if exc.fault == NoFault then
```

```
        ConsumeExcStackFrame(excReturn, psr.SPREALIGN);

    if HaveDSPExt() then
        APSR.GE        = psr.GE;
    if IsSecure() then
        CONTROL_S.SFPA = psr.SFPA;
    IPSR.Exception = psr.Exception;               // Load valid IPSR bits from memory
    EPSR.T         = psr.T;                       // Load valid EPSR bits from memory
    if HaveMainExt() then
        APSR<31:27>       = psr<31:27>;           // Load valid APSR bits from memory
        SetITSTATEAndCommit(psr.IT);              // Load valid ITSTATE from memory (also
            handles ICI and ECI)
    else
        APSR<31:28>       = psr<31:28>;           // Load valid APSR bits from memory
    return exc;
```

## E2.1.293 PreserveFPState

```
// PreserveFPState()
// =================

PreserveFPState()
    // Check if there is any lazy FP state to be preserved.
    isSecure = FPCCR_S.S == '1';
    lspact   = if isSecure then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
    if lspact == '0' then
        return;

    // Preserve FP state using address, privilege and relative
    // priorities recorded during original stacking. Derived
    // exceptions are handled by TakePreserveFPException().

    // The checks usually performed for stacking using ValidateAddress()
    // are performed, with the value of ExecutionPriority()
    // overridden by -1 if FPCCR.HFRDY == '0'.

    if isSecure then
        ispriv    = FPCCR_S.USER      == '0';
        splimviol = FPCCR_S.SPLIMVIOL == '1';
        fpcar     = FPCAR_S;
    else
        ispriv    = FPCCR_NS.USER      == '0';
        splimviol = FPCCR_NS.SPLIMVIOL == '1';
        fpcar     = FPCAR_NS;

    // Check if the background context had access to the FPU
    excInfo = CheckCPEnabled(10, ispriv, isSecure);

    // Only perform the memory accesses if the stack limit hasn't been violated
    bfExcInfo = DefaultExcInfo();
    if !splimviol && excInfo.fault == NoFault then
        // If IESB are enabled, barrier RAS / BusFault errors raised before Lazy FP stacking.
        // Because errors that are Synchronized at this point belong to the current context (
            the
        // context that executed the instruction that triggered the lazy stacking), and are
        // therefore handled normally and not by TakePreserveFPException.
        if AIRCR.IESB == '1' then
            HandleException(SynchronizeBusFault());

        // Whether these stores are interruptible is IMPLEMENTATION DEFINED.
        for i = 0 to 15
            if excInfo.fault == NoFault then
                addr    = fpcar + (4*i);
                excInfo = MemA_with_priv_security(addr,4,AccType_LAZYFP,ispriv,isSecure,TRUE,
                    S[i]);

        if excInfo.fault == NoFault then
            addr    = fpcar + 0x40;
```

```
            excInfo = MemA_with_priv_security(addr,4,AccType_LAZYFP,ispriv,isSecure,TRUE,
                FPSCR);
        if HaveMve() && excInfo.fault == NoFault then
            addr    = fpcar + 0x44;
            excInfo = MemA_with_priv_security(addr,4,AccType_LAZYFP,ispriv,isSecure,TRUE,VPR)
                ;

        if isSecure && FPCCR_S.TS == '1' then
            for i = 0 to 15
                if excInfo.fault == NoFault then
                    addr    = fpcar + (4*i) + 0x48;
                    excInfo = MemA_with_priv_security(addr,4,AccType_LAZYFP,ispriv,TRUE,
                                                      TRUE,S[i+16]);

        // If IESB are enabled, barrier RAS / BusFault errors raised during Lazy FP stacking
            to
        // the original context. If errors do occur, BFSR.LSPERR is set.
        if AIRCR.IESB == '1' then
            bfExcInfo = SynchronizeBusFault(AccType_LAZYFP);

    // Handle any faults that have occured
    termInst = FALSE;
    if excInfo.fault != NoFault then
        termInst = termInst || TakePreserveFPException(excInfo);
    if bfExcInfo.fault != NoFault then
        termInst = termInst || TakePreserveFPException(bfExcInfo);

    // If exception with sufficient priority to pre-empt current instruction execution is
        raised
    // during FP state preserve, then termInst will be true and execution of the current
    // instruction should be terminated by calling EndOfInstruction(). If the exception
        results
    // in a lockup state, termInst will also be true.
    if termInst then
        EndOfInstruction();
    else
        // In case of NoFault or, on successful return from TakePreserveFPException(), the
            current
        // instruction execution continues and FPCCR.LSPACT will be cleared.
        // NOTE: If the stores are interrupted, the register content and LSPACT remain
            unchanged.
        if isSecure then
            FPCCR_S.LSPACT = '0';
        else
            FPCCR_NS.LSPACT = '0';

        // If the FP state is being treated as Secure then the registers are zeroed
        InvalidateFPRegs(isSecure && FPCCR_S.TS == '1', isSecure && FPCCR_S.TS == '1');
```

### E2.1.294 ProcessorID

```
// ProcessorID
// ===========
// Returns an integer that uniquely identifies the executing PE in the system.

integer ProcessorID();
```

### E2.1.295 PushCalleeStack

```
// PushCalleeStack()
// =================

ExcInfo PushCalleeStack(boolean doTailChain, EXC_RETURN_Type excReturn)
    // allocate space of the correct stack. NOTE: If we are tail chaining we
    // look at excReturn instead of CONTROL.SPSEL to work out which stack to use,
    // as SPSEL can report the wrong stack in tail chaining cases
    if doTailChain then
```

```
        if excReturn.Mode == '0' then
            mode   = PEMode_Handler;
            spName = RNamesSP_Main_Secure;
        else
            mode   = PEMode_Thread;
            spName = if excReturn.SPSEL == '1' then RNamesSP_Process_Secure else
                RNamesSP_Main_Secure;
    else
        spName = LookUpSP();
        mode   = CurrentMode();

    // Calculate the address of the base of the callee frame
    bits(32) frameptr = _SP(spName) - 0x28;

    /* only the stack locations, not the store order, are architected */
    // Write out integrity signature
    integritySig = if HaveMveOrFPExt() then 0xFEFA125A<31:1> : excReturn.FType else 0
        xFEFA125B<31:0>;
    exc = Stack(frameptr, 0x0,  spName, mode, integritySig);
    // Stack callee registers
    if exc.fault == NoFault then exc = Stack(frameptr, 0x8,  spName, mode, R[4]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0xC,  spName, mode, R[5]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x10, spName, mode, R[6]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x14, spName, mode, R[7]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x18, spName, mode, R[8]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x1C, spName, mode, R[9]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x20, spName, mode, R[10]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x24, spName, mode, R[11]);

    // Update the stack pointer
    spExc = _SP(spName, TRUE, FALSE, frameptr);
    return MergeExcInfo(exc, spExc);
```

## E2.1.296 PushStack

```
// PushStack()
// ===========

(ExcInfo, EXC_RETURN_Type) PushStack(boolean commitState)
    constant integer intFrameSize      = 0x20;
    constant integer fpCallerFrameSize = 0x48;
    constant integer fpCalleeFrameSize = 0x40;

    boolean pushFPCallerFrame = HaveMveOrFPExt() && CONTROL.FPCA == '1';
    boolean pushFPCalleeFrame = pushFPCallerFrame && IsSecure() && FPCCR_S.TS == '1';

    integer framesize = intFrameSize;
    // In the case where a NOCP usage fault is generated, FP stack space is not allocated
    if IsSecure() || NSACR.CP10 == '1' then
        if pushFPCallerFrame then framesize = framesize + fpCallerFrameSize;
        if pushFPCalleeFrame then framesize = framesize + fpCalleeFrameSize;

    /* allocate space on the correct stack */
    bits(1) frameptralign;
    frameptralign = SP<2>;
    frameptr      = (SP - framesize) AND NOT(ZeroExtend('100',32));
    spName        = LookUpSP();

    // Prepare architecture state for stacking
    retState           = ReturnState(commitState);
    RETPSR_Type retpsr = XPSR<31:0>;
    retpsr.IT          = retState.ITState; // (also ICI and ECI); see ReturnState() in-line
        note for information on XPSR.IT bits
    retpsr.SPREALIGN   = frameptralign;
    retpsr.SFPA        = if IsSecure() then CONTROL_S.SFPA else '0';
    mode               = CurrentMode();

    /* only the stack locations, not the store order, are architected */
```

```
    exc                               = Stack(frameptr, 0x0,  spName, mode, R[0]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x4,  spName, mode, R[1]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x8,  spName, mode, R[2]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0xC,  spName, mode, R[3]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x10, spName, mode, R[12]);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x14, spName, mode, retState.LoopCount);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x18, spName, mode, retState.FetchAddr);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x1C, spName, mode, retpsr);
    frameOffset = intFrameSize;

    if pushFPCallerFrame then
        newExc = DefaultExcInfo();
        // LSPACT should not be active at the same time as CONTROL.FPCA. This
        // is a possible attack scenario so raise a SecureFault.
        lspact = if FPCCR_S.S == '1' then FPCCR_S.LSPACT else FPCCR_NS.LSPACT;
        if HaveSecurityExt() && lspact == '1' then
            SFSR.LSERR = '1';
            newExc     = CreateException(SecureFault);
        elsif !IsSecure() && NSACR.CP10 == '0' then
            UFSR_S.NOCP = '1';
            newExc      = CreateException(UsageFault, TRUE, TRUE);
        elsif FPCCR.LSPEN == '0' then
            if exc.fault == NoFault then
                exc = CheckCPEnabled(10);

            if exc.fault == NoFault then
                for i = 0 to 15
                    if exc.fault == NoFault then
                        exc = Stack(frameptr, frameOffset + (4*i), spName, mode, S[i]);
                if exc.fault == NoFault then
                    exc = Stack(frameptr, frameOffset + 0x40, spName, mode, FPSCR);
                if HaveMve() && exc.fault == NoFault then
                    exc = Stack(frameptr, frameOffset + 0x44, spName, mode, VPR);
                frameOffset = frameOffset + fpCallerFrameSize;

                if pushFPCalleeFrame then
                    for i = 0 to 15
                        if exc.fault == NoFault then
                            exc = Stack(frameptr, frameOffset + (4*i), spName, mode, S[i+16]);

            (cpEnabled, -) = IsCPEnabled(10);
            if cpEnabled then
                InvalidateFPRegs(pushFPCalleeFrame, pushFPCalleeFrame);
        else
            UpdateFPCCR(frameptr + frameOffset, TRUE);

        if newExc.fault != NoFault then
            // It is IMPLEMENTATION_DEFINED whether to drop the earlier MemFault
            // if the Secure fault or NOCP fault is also generated subsequently.
            // If MemFault is not dropped, it will be merged with Secure/NOCP fault
            // based on exception priority as per MergeExcInfo().
            if boolean IMPLEMENTATION_DEFINED "Drop previously generated exceptions" then
                exc = newExc;
            else
                exc = MergeExcInfo(exc, newExc);

    // Set the stack pointer to be at the bottom of the new stack frame
    spExc = _SP(spName, TRUE, FALSE, frameptr);
    exc   = MergeExcInfo(exc, spExc);

    // Some excReturn bits (eg ES, SPSEL) are set by ExceptionTaken
    EXC_RETURN_Type partialExcReturn = 0xFFFFFFF8<31:0>;
    partialExcReturn.S     = if IsSecure()            then '1'              else '0';
    partialExcReturn.FType = if HaveMveOrFPExt()      then NOT(CONTROL.FPCA) else '1';
    partialExcReturn.Mode  = if mode == PEMode_Thread then '1'              else '0';
    return (exc, partialExcReturn);
```

### E2.1.297 Q

```
// Q[] - non-assignment forms
// ==========================

bits(32) Q[integer idx, integer beat]
    assert idx  >= 0 && idx  <= 7;
    assert beat >= 0 && beat <= 3;
    return S[(idx * 4) + beat];


// Q[] - assignment forms
// ======================

Q[integer idx, integer beat] = bits(32) value
    assert idx  >= 0 && idx  <= 7;
    assert beat >= 0 && beat <= 3;
    S[(idx * 4) + beat] = value;
```

### E2.1.298 R

```
// R[]
// ===

// Non-assignment form

bits(32) R[integer n]
    return RName[LookUpRName(n)];

// Assignment form

R[integer n] = bits(32) value
    assert n != 15;
    RName[LookUpRName(n)] = value;
    return;
```

### E2.1.299 RaiseAsyncBusFault

```
// RaiseAsyncBusFault()
// ====================

RaiseAsyncBusFault()
    if HaveMainExt() then
        BFSR.IMPRECISERR = '1';

    // To make ensure errors are containable asnyc BusFault's escalate as if they were
    // synchronous if implicit error synchronisation barriers are enabled.
    handleSynchronously = AIRCR.IESB == '1';
    excInfo             = CreateException(BusFault, FALSE, IsSecure(), handleSynchronously);
    HandleException(excInfo);
```

### E2.1.300 RawExecutionPriority

```
// RawExecutionPriority()
// ======================
// Determine the current execution priority without the effect of priority boosting

integer RawExecutionPriority()
    execPri = HighestPri();
    for i = 2 to MaxExceptionNum() // IPSR values of the exception handlers
        for j = 0 to 1             // Check both Non-secure and Secure exceptions
            secure = (j == 0);
            if IsActiveForState(i, secure) then
                // PRIGROUP effect applied in ExceptionPriority
                effectivePriority = ExceptionPriority(i, secure, TRUE);
```

```
                if effectivePriority < execPri then
                    execPri = effectivePriority;
    assert execPri IN {-4 .. 256};
    return execPri;
```

### E2.1.301 replicate

```
// Replicate()
// ===========

bits(M*N) Replicate(bits(M) x, integer N);

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);
```

### E2.1.302 ResetRegs

```
// ResetRegs
// ============
// Sets all registers that have architecturally-defined reset
// values to those values

ResetRegs();
```

### E2.1.303 RestrictedNSPri

```
// RestrictedNSPri()
// =================
// The priority to which Non-secure exceptions are restricted if AIRCR.PRIS is set

integer RestrictedNSPri()
    return 0x80;
```

### E2.1.304 ReturnState

```
// ReturnState()
// =============

INSTR_EXEC_STATE_Type ReturnState(boolean commitState)

    // Whether the return address (and associated IT state) point to the current
    // instruction or the next instruction only depends on whether the
    // instruction executed correctly, and not the type of exception.
    //
    // For trivial cases this behavior matches the following expectation:-
    //  * Faults (eg MemManage, UsageFault, etc) result in the return address
    //    pointing to the instruction that caused the fault.
    //  * Interrupts and SVC's result in the return address pointing to the next
    //    instruction.
    //
    // However it is important to realise that the behavior can differ from the
    // expectation above in complex cases. The following examples illustrate how
    // and why the behavior can be different:-
    //  1)  A MemManage fault occurring at the same time as a higher priority
    //      interrupt. The interrupt is taken first due to its priority, but the
    //      return address is set to the current instruction because it didn't
    //      execute successfully. This ensures the return state is correct for
    //      when the pending MemManage fault is taken (which may occur by tail
    //      chaining after the interrupt handler returns).
    //  2)  The architecture states:-
    //          "A fault that is escalated to the priority of a HardFault
    //          retains the return address value of the original fault."
    //      So a SVC that escalates to a HardFault has the return address of the
```

```
    //      instruction after SVC (because the SVC succeeded is setting an
    //      exception pending).
    //  3)  The BusFault exception is disabled when a BusFault occurs during
    //      lazy FP state preservation. The fault remains pending until a store
    //      instruction re-enables the BusFault by writing to the SHCSR
    //      register, at which point the exception can be taken. However because
    //      the store instruction didn't cause the fault, it just allowed it to
    //      be taken the return address points to the instruction after the
    //      store.
    //
    //  NOTE: Asynchronous faults (eg async BusFault) deviate from this rule and
    //        have a return address set to the next instruction. Due to their
    //        asynchronous nature the address of the actual instruction that
    //        caused the fault is not known.
    //
    //        The return address is always halfword aligned, meaning bit<0> is
    //        always zero. If present the XPSR.IT bits saved to the stack are
    //        consistent with return address.
    if commitState then
        return GetInstrExecState(1);
    else
        return GetInstrExecState(0);
```

## E2.1.305 RName

```
// RName[] - assignment form
// =========================

RName[RNames reg] = bits(32) value
    case reg of
        when {RNamesSP_Main_NonSecure, RNamesSP_Process_NonSecure,
              RNamesSP_Main_Secure,    RNamesSP_Process_Secure}
            // It is IMPLEMENTATION DEFINED whether stack pointer limit checking
            // is performed for instructions that were previously UNPREDICTABLE
            // when modifying the stack pointer.
            applyLimit = boolean IMPLEMENTATION_DEFINED "SPLim check UNPRED instructions";
            exc = _SP(reg, FALSE, !applyLimit, value);
            assert applyLimit || exc.fault == NoFault;
        when RNamesPC
            // Direct PC writes not supported, PC updates must go through
            // LoadWritePC(), BranchReturn() or similar function
            assert FALSE;
        otherwise
            _RName[reg] = value;
    return;

// RName[] - non-assignment form
// =============================

bits(32) RName[RNames reg]
    bits(32) result;
    case reg of
        when {RNamesSP_Main_NonSecure, RNamesSP_Process_NonSecure,
              RNamesSP_Main_Secure,    RNamesSP_Process_Secure}
            result = _RName[reg]<31:2>:'00';
        when RNamesPC
            result = _RName[RNamesPC] + 4;
        otherwise
            result = _RName[reg];
    return result;
```

## E2.1.306 RNames

```
// The names of the core registers. SP is a Banked register.

enumeration RNames {RNames0, RNames1, RNames2, RNames3, RNames4, RNames5, RNames6,
                    RNames7, RNames8, RNames9, RNames10, RNames11, RNames12,
```

```
                      RNamesSP_Main_Secure, RNamesSP_Main_NonSecure,
                      RNamesLR, RNamesPC,
                      RNamesSP_Process_NonSecure, RNamesSP_Process_Secure};
```

### E2.1.307 ROR

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

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

### E2.1.308 ROR_C

```
// 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);
```

### E2.1.309 roundDown

```
// RoundDown()
// ===========

integer RoundDown(real x);
```

### E2.1.310 roundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x > 0.0 then RoundDown(x) else RoundUp(x);
```

### E2.1.311 roundUp

```
// RoundUp()
// =========

integer RoundUp(real x);
```

### E2.1.312 RRX

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

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

### E2.1.313 RRX_C

```
// 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);
```

### E2.1.314 RSPCheck

```
// RSPCheck[] - assignment form
// ============================

RSPCheck[integer n] = bits(32) value
    if n == 13 then
        - = _SP(LookUpSP(), FALSE, FALSE, value);
    else
        R[n] = value;
    return;
```

### E2.1.315 RZ

```
// RZ[] -- Read R15 as zero
// ========================

bits(32) RZ[integer n]
    assert n >= 0 && n <= 15;
    if n == 15 then
        return Zeros(32);
    else
        return R[n];
```

### E2.1.316 S

```
// S[]
// ===

// Non-assignment form

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    return _S[n];

// Assignment form

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    _S[n] = value;
    return;
```

### E2.1.317 Sat

```
// Sat()
// =====

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

### E2.1.318 SatQ

```
// 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);
```

### E2.1.319 SAttributes

```
// Security attributes associated with an address

type SAttributes is (
    boolean nsc,           // Non-secure callability of an address. FALSE = not
                           // callable from the Non-secure state
    boolean ns,            // Security of an address FALSE = Secure, TRUE = Non-secure
    bits(8) sregion,       // The SAU region number
    boolean srvalid,       // Set to 1 if the SAU region number is valid
    bits(8) iregion,       // The IDAU region number
    boolean irvalid        // Set to 1 if the IDAU region number is valid
)
```

### E2.1.320 SCS_UpdateStatusRegs

```
// SCS_UpdateStatusRegs()
// ======================
// Update status registers in the System Control Space (SCS)

SCS_UpdateStatusRegs();
```

### E2.1.321 SecureDebugMonitorAllowed

```
// SecureDebugMonitorAllowed()
// ===========================

boolean SecureDebugMonitorAllowed()
    if DAUTHCTRL_S.FSDMA == '1' then
        return TRUE;
    elsif DAUTHCTRL_S.SPIDENSEL == '1' then
        return DAUTHCTRL_S.INTSPIDEN == '1';
    else
        return ExternalSecureSelfHostedDebugEnabled();
```

### E2.1.322 SecureHaltingDebugAllowed

```
// SecureHaltingDebugAllowed()
// ===========================

boolean SecureHaltingDebugAllowed()
    if HaltingDebugAllowed() == FALSE then
        return FALSE;
    elsif DAUTHCTRL_S.SPIDENSEL == '1' then
        return DAUTHCTRL_S.INTSPIDEN == '1';
    else
        return ExternalSecureInvasiveDebugEnabled();
```

### E2.1.323 SecureNoninvasiveDebugAllowed

```
// SecureNoninvasiveDebugAllowed()
// ===============================

boolean SecureNoninvasiveDebugAllowed()
    if DHCSR.S_SDE == '1' then
        return TRUE;
```

```
    elsif !NoninvasiveDebugAllowed() then
        return FALSE;
    elsif DAUTHCTRL_S.SPNIDENSEL == '1' then
        return DAUTHCTRL_S.INTSPNIDEN == '1';
    else
        return ExternalSecureNoninvasiveDebugEnabled();
```

### E2.1.324 SecurityCheck

```
// SecurityCheck()
// ===============

SAttributes SecurityCheck(bits(32) address, boolean isinstrfetch, boolean isSecure)
    SAttributes result;
    addr = UInt(address);

    // Setup default attributes
    result.ns      = !HaveSecurityExt();
    result.nsc     = FALSE;
    result.sregion = Zeros(8);
    result.srvalid = FALSE;
    result.iregion = Zeros(8);
    result.irvalid = FALSE;
    idauExempt     = FALSE;
    idauNs         = TRUE;
    idauNsc        = TRUE;

    // If an IMPLEMENTATION DEFINED memory security attribution unit is present
    // query it and override defaults set above. The IDAU is subject to the same
    // 32byte minimum region granularity as the SAU/MPU.
    // NOTE: The defaults above are set such that the IDAU has no effect on the
    //       SAU.
    if boolean IMPLEMENTATION_DEFINED "IDAU present" then
        (idauExempt,
         idauNs,
         idauNsc,
         result.iregion,
         result.irvalid) = IDAUCheck(address<31:5>:'00000');

    // The 0xF0000000 -> 0xFFFFFFFF is always Secure for instruction fetches
    if isinstrfetch && (address<31:28> == '1111') then
        // Use default attributes defined above

    // Check if the address is exempt from SAU/IDAU checking.
    elsif idauExempt                                        ||   // IDAU specified exemption
          (isinstrfetch && (address<31:28> == '1110'))      ||   // Whole 0xExxxxxxx range
              exempt for IFetch
          ((addr >= 0xE0000000) && (addr <= 0xE0003FFF))    ||   // ITM, DWT, FPB, PMU
          ((addr >= 0xE0005000) && (addr <= 0xE0005FFF))    ||   // RAS error record registers
          ((addr >= 0xE000E000) && (addr <= 0xE000EFFF))    ||   // SCS
          ((addr >= 0xE002E000) && (addr <= 0xE002EFFF))    ||   // SCS NS alias
          ((addr >= 0xE0040000) && (addr <= 0xE0041FFF))    ||   // TPIU, ETM
          ((addr >= 0xE00FF000) && (addr <= 0xE00FFFFF)) then // ROM table
        // memory security reported as NS-Req, and no region information is supplied.
        result.ns      = !isSecure;
        result.irvalid = FALSE;

    else
        // If the SAU is enabled check its regions
        if SAU_CTRL.ENABLE == '1' then
            boolean multiRegionHit = FALSE;
            for r = 0 to (UInt(SAU_TYPE.SREGION) - 1)
                if SAU_REGION[r].ENABLE == '1' then
                    // SAU region enabled so perform checks
                    bits(32) base_address  = SAU_REGION[r].BADDR:'00000';
                    bits(32) limit_address = SAU_REGION[r].LADDR:'11111';
                    if ((UInt(base_address)  <= addr) &&
                        (UInt(limit_address) >= addr)) then
```

```
                        if result.srvalid then
                            multiRegionHit = TRUE;
                        else
                            result.ns      = SAU_REGION[r].NSC == '0';
                            result.nsc     = SAU_REGION[r].NSC == '1';
                            result.srvalid = TRUE;
                            result.sregion = r<7:0>;

            // If multiple regions are hit then report memory as Secure and not
            // Non-secure callable. Also don't report any region number
            // information.
            if multiRegionHit then
                result.ns      = FALSE;
                result.nsc     = FALSE;
                result.sregion = Zeros(8);
                result.srvalid = FALSE;

        // SAU disabled, check if whole address space should be marked as
        // Non-secure
        elsif SAU_CTRL.ALLNS == '1' then
            result.ns = TRUE;

        // Override the internal setting if the external attribution unit
        // reports more restrictive attributes.
        if !idauNs then
            if result.ns || (!idauNsc && result.nsc) then
                result.ns  = FALSE;
                result.nsc = idauNsc;

    return result;
```

### E2.1.325 SecurityState

```
// Type and definition of the current Security state of PE

enumeration SecurityState {SecurityState_NonSecure, SecurityState_Secure};
SecurityState CurrentState;
```

### E2.1.326 SendEvent

```
// SendEvent
// =========
// Performs a send event by setting the Event Register of every PE in multiprocessor system

SendEvent();
```

### E2.1.327 SerializeVFP

```
// SerializeVFP
// ============
// Ensures that any exceptional conditions in previous floating-point
// instructions have been detected

SerializeVFP();
```

### E2.1.328 SetActive

```
// SetActive()
// ===========

SetActive(integer exception, boolean isSecure, boolean setNotClear)
    if !HaveSecurityExt() then
        isSecure = FALSE;
    // If the exception target state is configurable there is only one active
```

```
    // bit. To represent this the Non-secure and Secure instances of the active
    // flags in the array are always set to the same value.
    if IsExceptionTargetConfigurable(exception) then
        if ExceptionTargetsSecure(exception, boolean UNKNOWN) == isSecure then
            ExceptionActive[exception] = if setNotClear then '11' else '00';
    else
        idx = if isSecure then 0 else 1;
        ExceptionActive[exception]<idx> = if setNotClear then '1' else '0';
```

### E2.1.329 SetDWTDebugEvent

```
// SetDWTDebugEvent()
// ==================
// Set a pending debug event to the PE

boolean SetDWTDebugEvent(boolean secure_match)
    if CanHaltOnEvent(secure_match) then
        DHCSR.C_HALT = '1';
        DFSR.DWTTRAP = '1';
        return TRUE;

    elsif HaveMainExt() && CanPendMonitorOnEvent(secure_match, TRUE, TRUE) then
        DEMCR.MON_PEND = '1';
        DFSR.DWTTRAP = '1';
        return TRUE;

    else
        return FALSE;
```

### E2.1.330 SetEventRegister

```
// SetEventRegister()
// ==================
// Set the Event Register of the current PE

SetEventRegister();
```

### E2.1.331 SetExclusiveMonitors

```
// SetExclusiveMonitors()
// ======================

SetExclusiveMonitors(bits(32) address, integer size)
    boolean isSecure = CurrentState == SecurityState_Secure;
    (excInfo, memaddrdesc) = ValidateAddress(address, AccType_NORMAL, FindPriv(),
                                             isSecure, FALSE, TRUE);
    HandleException(excInfo);

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
```

### E2.1.332 SetITSTATEAndCommit

```
// SetITSTATEAndCommit()
// =====================

SetITSTATEAndCommit(ITSTATEType it)
    // This function directly commits the change to the ITSTATE, so ThisInstrITSTATE()
    // and NextInstrITSTATE() both point to the target address.
    _NextInstrITState = it;
    _ITStateChanged   = TRUE;
    EPSR.IT           = it;
    return;
```

### E2.1.333 SetPending

```
// SetPending()
// ============

SetPending(integer exception, boolean isSecure, boolean setNotClear)
    if !HaveSecurityExt() then
        isSecure = FALSE;
    // If the exception target state is configurable there is only one pending
    // bit. To represent this, the Non-secure and Secure instances of the pending
    // flags in the array are always set to the same value.
    if IsExceptionTargetConfigurable(exception) then
        ExceptionPending[exception] = if setNotClear then '11' else '00';
    else
        idx = if isSecure then 0 else 1;
        ExceptionPending[exception]<idx> = if setNotClear then '1' else '0';
```

### E2.1.334 SetThisInstrDetails

```
// SetThisInstrDetails
// ===================

SetThisInstrDetails(bits(32) opcode, integer len)
    // Insert the instruction into the queue at the first free slot. For
    // instruction with no beat behaviour this should always be the first slot.
    // NOTE: MVE instructions in IT blocks do not have beat wise execution.
    i          = 0;
    isBeatInst = IsMveBeatWiseInstruction(opcode) && !InITBlock();
    repeat
        emptySlot = !_InstInfo[i].Valid;
        if emptySlot && (isBeatInst || i == 0) then
            _InstInfo[i].Valid  = TRUE;
            _InstInfo[i].Length = len;
            _InstInfo[i].Opcode = opcode;
        i = i + 1;
    until emptySlot || (!isBeatInst && i > 0) || (i >= MAX_OVERLAPPING_INSTRS);
```

### E2.1.335 SetThisInstrDetails

```
// SetThisInstrDetails
// ===================
// Set the details of current instruction

SetThisInstrDetails(bits(32) opcode, integer len);
```

### E2.1.336 SetVPTMask

```
// SetVPTMask()
// ============

SetVPTMask(integer beat, bits(4) mask)
    // Only one mask field is available for each pair of beats.
    assert beat<0> == '1';
    Elem[VPR<23:16>, beat DIV 2, 4] = mask;
    // Since the mask has been modified don't advance the VPT state after this
    // instruction beat.
    _AdvanceVPTState = FALSE;
```

### E2.1.337 Shift

```
// Shift()
// =======

```

```
bits(N) Shift(bits(N) value, SRType sr_type, integer amount, bit carry_in)
    (result, -) = Shift_C(value, sr_type, amount, carry_in);
    return result;
```

### E2.1.338 Shift_C

```
// Shift_C()
// =========

(bits(N), bit) Shift_C(bits(N) value, SRType sr_type, integer amount, bit carry_in)
    assert !(sr_type == SRType_RRX && amount != 1);

    if amount == 0 then
        (result, carry_out) = (value, carry_in);
    else
        case sr_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);
```

### E2.1.339 SignedSat

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

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

### E2.1.340 SignedSatQ

```
// 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);
```

### E2.1.341 signExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

### E2.1.342 Sleeping

```
// Indicates the PE is sleeping

boolean Sleeping;
```

### E2.1.343 SleepOnExit

```
// SleepOnExit()
// =============
// Optionally returns PE to a power-saving mode on return from the only
// active exception

SleepOnExit();
```

### E2.1.344 SP

```
// SP
// ==

// Non-assignment form

bits(32) SP
    return R[13];

// Assignment form

SP = bits(32) value
    RSPCheck[13] = value;
```

### E2.1.345 SP_Main

```
// SP_Main
// =======

// Non-assignment form

bits(32) SP_Main
    value = if IsSecure() then SP_Main_Secure else SP_Main_NonSecure;
    return value;

// Assignment form

SP_Main = bits(32) value
    if IsSecure() then
        SP_Main_Secure    = value;
    else
        SP_Main_NonSecure = value;
```

### E2.1.346 SP_Main_NonSecure

```
// SP_Main_NonSecure
// =================

// Non-assignment form

bits(32) SP_Main_NonSecure
    return _SP(RNamesSP_Main_NonSecure);

// Assignment form

SP_Main_NonSecure = bits(32) value
    - = _SP(RNamesSP_Main_NonSecure, FALSE, FALSE, value);
```

### E2.1.347 SP_Main_Secure

```
// SP_Main_Secure
// ==============

// Non-assignment form

bits(32) SP_Main_Secure
    return _SP(RNamesSP_Main_Secure);

// Assignment form

SP_Main_Secure = bits(32) value
    - = _SP(RNamesSP_Main_Secure, FALSE, FALSE, value);
```

### E2.1.348 SP_Process

```
// SP_Process
// ==========

// Non-assignment form

bits(32) SP_Process
    value = if IsSecure()
            then SP_Process_Secure else SP_Process_NonSecure;
    return value;

// Assignment form

SP_Process = bits(32) value
    if IsSecure() then
        SP_Process_Secure    = value;
    else
        SP_Process_NonSecure = value;
```

### E2.1.349 SP_Process_NonSecure

```
// SP_Process_NonSecure
// ====================

// Non-assignment form

bits(32) SP_Process_NonSecure
    return _SP(RNamesSP_Process_NonSecure);

// Assignment form

SP_Process_NonSecure = bits(32) value
    - = _SP(RNamesSP_Process_NonSecure, FALSE, FALSE, value);
```

### E2.1.350 SP_Process_Secure

```
// SP_Process_Secure
// =================

// Non-assignment form

bits(32) SP_Process_Secure
    return _SP(RNamesSP_Process_Secure);

// Assignment form

SP_Process_Secure = bits(32) value
    - = _SP(RNamesSP_Process_Secure, FALSE, FALSE, value);
```

### E2.1.351 SpeculativeSynchronizationBarrier

```
// Speculative Synchronisation Barrier
// ===================================
// Perform a Speculative Synchronization Barrier

SpeculativeSynchronizationBarrier();
```

### E2.1.352 SRType

```
// Different types of shift and rotate operations

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

### E2.1.353 Stack

```
// Stack
// =====

// Assignment form

ExcInfo Stack(bits(32) frameptr, integer offset, RNames spreg, PEMode mode, bits(32) value)
    // This function is used to perform register stacking operations that are
    // done around exception handling. If the stack pointer is below the stack
    // pointer limit but the access itself is above the limit it is
    // IMPLEMENTATION DEFINED whether the write is performed. If the
    // address of access is below the limit the access is not performed
    // regardless of the stack pointer value.
    if !ViolatesSPLim(LookUpSP(), frameptr) then
        doAccess = TRUE;
    else
        doAccess = boolean IMPLEMENTATION_DEFINED "Push non-violating locations";

    address = frameptr + offset;
    if doAccess && !ViolatesSPLim(LookUpSP(), address) then
        secure = ((spreg == RNamesSP_Main_Secure)   ||
                  (spreg == RNamesSP_Process_Secure));
        // Work out if the stack operations should be privileged or not
        if secure then
            isPriv = CONTROL_S.nPRIV == '0';
        else
            isPriv = CONTROL_NS.nPRIV == '0';
        isPriv = isPriv || (mode == PEMode_Handler);
        // Finally perform the memory operations
        excInfo = MemA_with_priv_security(address,4,AccType_STACK,isPriv,secure,TRUE,value);
    else
        excInfo = DefaultExcInfo();
    return excInfo;

// Non-assignment form

(ExcInfo, bits(32)) Stack(bits(32) frameptr, integer offset, RNames spreg, PEMode mode)
    secure = ((spreg == RNamesSP_Main_Secure)   ||
              (spreg == RNamesSP_Process_Secure));
    // Work out if the stack operations should be privileged or not
    if secure then
        isPriv = CONTROL_S.nPRIV == '0';
    else
        isPriv = CONTROL_NS.nPRIV == '0';
    isPriv = isPriv || (mode == PEMode_Handler);
    // Finally perform the memory operations
    address = frameptr + offset;
    (excInfo, value) = MemA_with_priv_security(address,4,AccType_STACK,isPriv,secure,TRUE);
    return (excInfo, value);
```

### E2.1.354 StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

FPSCR_Type StandardFPSCRValue()
    return '00000' : FPSCR.AHP : '1100000000000000000000000';
```

### E2.1.355 SteppingDebug

```
// SteppingDebug()
// ===============

SteppingDebug()
    // Process step requests, stepping must be avoided if a pending event is
    // already in flight.
    if CanHaltOnEvent(IsSecure()) && DHCSR.C_STEP == '1' && DHCSR.C_HALT == '0' then
        // If C_STEP is set then pend a debug halt for the next instruction.
        DHCSR.C_HALT    = '1';
        DFSR.HALTED     = '1';
    elsif CanPendMonitorOnEvent(IsSecure(), TRUE, TRUE) && DEMCR.MON_STEP == '1' && DEMCR.
        MON_PEND == '0' then
        // If MON_STEP is set then pend the an exception for the next instruction.
        DEMCR.MON_PEND = '1';
        DFSR.HALTED    = '1';
```

### E2.1.356 SynchronizeBusFault

```
// SynchronizeBusFault()
// =====================

ExcInfo SynchronizeBusFault()
    return SynchronizeBusFault(FALSE);

ExcInfo SynchronizeBusFault(AccType acctype)
    return SynchronizeBusFault(acctype == AccType_LAZYFP);

ExcInfo SynchronizeBusFault(boolean isLazyStatePreservation)
    // Force any latent BusFaults to be recognised
    faultDetected = BusFaultBarrier();
    if faultDetected then
        if isLazyStatePreservation then
            BFSR.LSPERR      = '1';
        else
            BFSR.IMPRECISERR = '1';
        // To ensure errors are containable, asnyc BusFaults escalate as if they were
        // synchronous if implicit error synchronisation barriers are enabled.
        handleSynchronously = AIRCR.IESB == '1';
        excInfo             = CreateException(BusFault, FALSE, IsSecure(),
            handleSynchronously);
    else
        excInfo = DefaultExcInfo();
    return excInfo;
```

### E2.1.357 T32ExpandImm

```
// T32ExpandImm()
// ==============

bits(32) T32ExpandImm(bits(12) imm12)

    // APSR.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm_C(imm12, APSR.C);

    return imm32;
```

### E2.1.358 T32ExpandImm_C

```
// T32ExpandImm_C()
// ================

(bits(32), bit) T32ExpandImm_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);
```

### E2.1.359 TailChain

```
// TailChain()
// ===========

(ExcInfo, EXC_RETURN_Type) TailChain(integer exceptionNumber, boolean excIsSecure,
    EXC_RETURN_Type excReturn)
    // Refresh LR with the excReturn value, ready for the next exception
    if !HaveMveOrFPExt() then
        excReturn.FType = '1';
    excReturn.PREFIX = Ones(8);

    return ExceptionTaken(exceptionNumber, TRUE, excIsSecure, IgnoreFaults_NONE, excReturn);
```

### E2.1.360 TakePreserveFPException

```
// TakePreserveFPException()
// =========================

boolean TakePreserveFPException(ExcInfo excInfo)
    assert HaveMveOrFPExt();
    assert excInfo.origFault IN {DebugMonitor, SecureFault, MemManage, BusFault, UsageFault};

    // Get the details of the original fault so that any escalation to HardFault / Lockup
        based
    // on the current execution priority is ignored. Escalation is performed manually against
    // the FPCCR.*RDP fields below.
    exception = excInfo.origFault;
    isSecure  = excInfo.origFaultIsSecure;
    fpccr     = if isSecure then FPCCR_S else FPCCR_NS;

    if FPCCR_S.MONRDY == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
    if FPCCR_S.BFRDY  == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
    if FPCCR_S.SFRDY  == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
    if fpccr.UFRDY    == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
    if fpccr.MMRDY    == '1' && FPCCR_S.HFRDY == '0' then UNPREDICTABLE;
```

```
    if exception == DebugMonitor && FPCCR_S.MONRDY == '0' then
        // ignore DebugMonitor exception
        return FALSE;

    // Handle exception specific details like escalation and syndrome information
    case exception of
        when MemManage
            escalate = fpccr.MMRDY == '0';
        when UsageFault
            escalate = fpccr.UFRDY == '0';
        when BusFault
            escalate = FPCCR_S.BFRDY == '0';
        when SecureFault
            escalate = FPCCR_S.SFRDY == '0';
        otherwise
            escalate = FALSE;
    if escalate then
        exception = HardFault;
        // Faults that originally targeted the Secure state still target the
        // Secure state even if HardFault normally targets Non-secure.
        isSecure  = isSecure || ExceptionTargetsSecure(HardFault, isSecure);

    // Check if the exception is enabled and has sufficient priority to
    // preempt and be taken straight away.
    termInst = FALSE;
    if (ExceptionPriority(exception, isSecure, TRUE) < ExecutionPriority()) &&
        ExceptionEnabled(exception, isSecure) then
        if escalate then
            HFSR.FORCED = '1';
        // Set the exception pending and terminate the current instruction. This
        // leaves FP disabled (that is CONTROL.FPCA set to 0) and prevents the
        // preempting exception entry reserving space for a redundant FP state.
        SetPending(exception, isSecure, TRUE);
        termInst = TRUE;
    else
        // If the reason the exception cannot preempt is because of the fact that
        // HardFault couldn't be entered by the context the FP state belongs to
        // then enter the lockup state.
        if FPCCR_S.HFRDY == '0' then
            Lockup(FALSE); // Lockup at current priority, lock-up address = 0xEFFFFFFE
            termInst = TRUE;
        else
            if escalate then
                HFSR.FORCED = '1';
            // Set the exception pending so it will be taken after the current
            // handler returns.
            SetPending(exception, isSecure, TRUE);
    return termInst;
```

## E2.1.361 TakeReset

```
// TakeReset()
// ===========

TakeReset()
    // If the Security Extension is implemented the PE resets into Secure state.
    // If the Security Extension is not implemented the PE resets into Non-secure state.
    CurrentState = if HaveSecurityExt() then SecurityState_Secure else
        SecurityState_NonSecure;

    ResetRegs();                        // Catch-all function for System Control Space reset
    if HaveMainExt() then
        LR = Ones(32);                  // Preset to an illegal exception return value
        SetITSTATEAndCommit(Zeros(8));  // IT/ICI bits cleared
    else
        LR = bits(32) UNKNOWN;          // Value must be initialised by software

    for i = 0 to MAX_OVERLAPPING_INSTRS-1
```

```
        _InstInfo[i].Valid = FALSE;
    if HaveMve() then
        VPR = bits(32) UNKNOWN;

    // Reset internal run state
    Halted   = FALSE;
    Sleeping = FALSE;
    LockedUp = FALSE;

    // Initialize the Floating Point Extn
    if HaveMveOrFPExt() then
        for i = 0 to 31
            S[i] = bits(32) UNKNOWN;

    for i = 0 to MaxExceptionNum()          // All exceptions Inactive
        ExceptionActive[i] = '00';
    ClearExclusiveLocal(ProcessorID());     // Synchronization (LDREX* / STREX*) monitor
        support
    ClearEventRegister();                   // See WFE instruction for more information
    for i = 0 to 12
        R[i] = bits(32) UNKNOWN;

    // Clearing stack limit registers
    if HaveMainExt() then
        MSPLIM_NS = Zeros(32);
        PSPLIM_NS = Zeros(32);
    if HaveSecurityExt() then
        MSPLIM_S  = Zeros(32);
        PSPLIM_S  = Zeros(32);

    // Load the initial value of the stack pointer and the reset value from the
    // vector table. The order of the loads is IMPLEMENTATION DEFINED
    (excSp,  sp)    = Vector[0,     HaveSecurityExt()];
    (excRst, start) = Vector[Reset, HaveSecurityExt()];
    if excSp.fault != NoFault || excRst.fault != NoFault then
        Lockup(TRUE);

    // Initialize the stack pointers and start execution at the reset vector
    if HaveSecurityExt() then
        SP_Main_Secure    = sp;
        SP_Main_NonSecure = ((bits(30) UNKNOWN):'00');
        SP_Process_Secure = ((bits(30) UNKNOWN):'00');
    else
        SP_Main_NonSecure = sp;
    SP_Process_NonSecure  = ((bits(30) UNKNOWN):'00');
    EPSR.T = start<0>;
    BranchTo(start, TRUE);

    // Trigger a debug event even if resetting into secure state
    if DHCSR.C_DEBUGEN == '1' && DEMCR.VC_CORERESET == '1' &&
       (HasArchVersion(Armv8p1) || CanHaltOnEvent(HaveSecurityExt())) then
        DHCSR.C_HALT = '1';
        DFSR.VCATCH  = '1';
```

## E2.1.362 ThisInstr

```
// ThisInstr()
// ===========

bits(32) ThisInstr()
    return ThisInstr(if HaveMve() then _InstID else 0);

bits(32) ThisInstr(integer instID)
    if !_InstInfo[instID].Valid then
        return bits(32) UNKNOWN;
    return _InstInfo[instID].Opcode;
```

### E2.1.363 ThisInstrAddr

```
// ThisInstrAddr()
// ===============

bits(32) ThisInstrAddr()
    return _CurrentInstrExecState.FetchAddr;
```

### E2.1.364 ThisInstrITState

```
// ThisInstrITState()
// ==================

ITSTATEType ThisInstrITState()
    if HaveMainExt() then
        value = _CurrentInstrExecState.ITState;
    else
        value = Zeros(8);
    return value;
```

### E2.1.365 ThisInstrLength

```
// ThisInstrLength()
// =================

integer ThisInstrLength()
    return ThisInstrLength(if HaveMve() then _InstID else 0);

integer ThisInstrLength(integer instID)
    if !_InstInfo[instID].Valid then
        return 0;
    return _InstInfo[instID].Length;
```

### E2.1.366 TopLevel

```
// TopLevel()
// ==========

// This function is called one time for each tick the PE is not in a sleep
// state. It handles all instruction processing, including fetching the opcode,
// decode and execute. It also handles pausing execution when in the lockup
// state.
TopLevel()
    // Process any pending reset.
    if AIRCR_S.SYSRESETREQ == '1' then
        TakeReset();

    UpdateDebugEnable();

    // Should state of the current instruction (or oldest beat) be committed
    commitState = FALSE;

    // If the PE is halted then do nothing, otherwise process the next
    // instruction.
    if !IsDebugState() then
        // If not locked up, process the next instruction, or just the in flight
        // beats if a halt or monitor exception is pending.
        if !LockedUp then
            commitState = InstructionExecute(!PendingDebugHalt() &&
                                             !PendingDebugMonitor());

        // If a debug halt was requested and there are no active vector chains
        // then halt if allowed to do so.
        if !InstructionsInFlight() && PendingDebugHalt() then
            Halt();
```

```
31
32          // Process any debug step requests
33          SteppingDebug();
34
35      elsif DHCSR.C_HALT == '0' then
36          // Resume the PE if a resume from debug halt was requested.
37          Halted             = FALSE;
38          DHCSR.S_RESTART_ST = '1';
39
40      if !IsDebugState() then
41          try
42              // Process and take any pending exceptions.
43              if HandleExceptionTransitions(commitState) then
44                  // If an exception has been taken, process any step request now,
45                  // not on the next instruction
46                  SteppingDebug();
47
48                  // Pend vector catch debug state when needed
49                  VectorCatchDebug();
50
51              // If the PC has moved away from the lockup address (eg because an
52              // NMI has been taken) leave the lockup state.
53              if LockedUp && NextInstrAddr() != 0xEFFFFFFE<31:0> then
54                  LockedUp = FALSE;
55
56              // Advance the PC and commit instruction state as necessary if not
57              // locked up.
58              if !LockedUp then
59                  InstructionAdvance(commitState);
60          catch exn
61              // Do not catch UNPREDICTABLE or internal errors.
62              when IsExceptionTaken(exn)
63                  // The correct architectural behavior for any exceptions is
64                  // performed inside HandleExceptionTransitions. So no
65                  // additional actions are required in this catch block.
```

## E2.1.367 TTResp

```
1  // TTResp()
2  // ========
3
4  bits(32) TTResp(bits(32) address, boolean alt, boolean forceunpriv)
5      TT_RESP_Type resp = Zeros();
6
7      // Only allow security checks if currently in Secure state
8      if IsSecure() then
9          sAttributes = SecurityCheck(address, FALSE, IsSecure());
10         if sAttributes.srvalid then
11             resp.SREGION = sAttributes.sregion;
12             resp.SRVALID = '1';
13         if sAttributes.irvalid then
14             resp.IREGION = sAttributes.iregion;
15             resp.IRVALID = '1';
16         addrSecure = if sAttributes.ns  then '0' else '1';
17         resp.S     = addrSecure;
18
19     // MPU region information only available when privileged or when
20     // inspecting the other MPU state.
21     other_domain  = (alt != IsSecure());
22     if CurrentModeIsPrivileged() || alt then
23         (write, read, region, hit) = IsAccessible(address, forceunpriv, other_domain);
24         if hit then
25             resp.MREGION = region;
26             resp.MRVALID = '1';
27         resp.R  = read;
28         resp.RW = write;
29         if IsSecure() then
30             resp.NSR  = read  AND NOT addrSecure;
```

```
            resp.NSRW = write AND NOT addrSecure;

    return resp;
```

### E2.1.368 UnprivHaltingDebugAllowed

```
// UnprivHaltingDebugAllowed()
// ===========================

boolean UnprivHaltingDebugAllowed(boolean isSecure)
    allowed = DAUTHCTRL_S.UIDEN  == '1';
    if !isSecure then
        // Secure unprivileged debug also grants Nonsecure unprivileged debug.
        allowed = allowed || DAUTHCTRL_NS.UIDEN == '1';
    return HaveUDE() && allowed && !CurrentModeIsPrivileged(isSecure);
```

### E2.1.369 UnsignedSat

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

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

### E2.1.370 UnsignedSatQ

```
// 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);
```

### E2.1.371 UpdateDebugEnable

```
// UpdateDebugEnable()
// ===================
// Update DHCSR.S_SDE, DEMCR.SDME, and unprivileged debug enables for each instruction

UpdateDebugEnable()
    // DHCSR.S_SDE and unprivileged debug enables are frozen if the PE is in Debug state
    if !Halted then
        nsUide         = UnprivHaltingDebugAllowed(FALSE);
        sUide          = UnprivHaltingDebugAllowed(TRUE);
        DHCSR.S_SDE    = if sUide  ||  SecureHaltingDebugAllowed() then '1' else '0';
        DHCSR.S_SUIDE  = if sUide  && !SecureHaltingDebugAllowed() then '1' else '0';
        DHCSR.S_NSUIDE = if nsUide && !HaltingDebugAllowed()       then '1' else '0';

    // DEMCR.SDME is frozen if DebugMonitor is active or pending
    if HaveDebugMonitor() && ExceptionActive[DebugMonitor] == '00' && DEMCR.MON_PEND == '0'
        then
        DEMCR.SDME = if SecureDebugMonitorAllowed() then '1' else '0';
```

### E2.1.372 UpdateFPCCR

```
// UpdateFPCCR()
// =============

UpdateFPCCR(bits(32) frameptr, boolean applySpLim)
    assert(HaveMveOrFPExt());

    FPCAR.ADDRESS = frameptr<31:3>;
    // Flag if the context address violates the stack pointer limit. If the
    // limit has been violated PreserveFPState() will zero the registers if
    // required, but will not save the context to the stack.
    if applySpLim && ViolatesSPLim(LookUpSP(), frameptr) then
        FPCCR.SPLIMVIOL = '1';
    else
        FPCCR.SPLIMVIOL = '0';
    FPCCR.LSPACT = '1';

    execPri  = ExecutionPriority();
    isSecure = IsSecure();
    FPCCR_S.S = if isSecure then '1' else '0';
    if CurrentModeIsPrivileged() then
        FPCCR.USER = '0';
    else
        FPCCR.USER = '1';
    if CurrentMode() == PEMode_Thread then
        FPCCR.THREAD = '1';
    else
        FPCCR.THREAD = '0';
    if execPri > -1 then
        FPCCR_S.HFRDY = '1';
    else
        FPCCR_S.HFRDY = '0';
    targetSecure = AIRCR.BFHFNMINS == '0';
    busfaultpri = ExceptionPriority(BusFault, targetSecure, FALSE);
    if SHCSR_S.BUSFAULTENA == '1' && execPri > busfaultpri then
        FPCCR_S.BFRDY = '1';
    else
        FPCCR_S.BFRDY = '0';
    memfaultpri = ExceptionPriority(MemManage, isSecure, FALSE);
    if SHCSR.MEMFAULTENA == '1' && execPri > memfaultpri then
        FPCCR.MMRDY = '1';
    else
        FPCCR.MMRDY = '0';
    usagefaultpri = ExceptionPriority(UsageFault, FALSE, FALSE);
    if SHCSR_NS.USGFAULTENA == '1' && execPri > usagefaultpri then
        FPCCR_NS.UFRDY = '1';
    else
        FPCCR_NS.UFRDY = '0';
    usagefaultpri = ExceptionPriority(UsageFault, TRUE, FALSE);
    if SHCSR_S.USGFAULTENA == '1' && execPri > usagefaultpri then
        FPCCR_S.UFRDY = '1';
    else
        FPCCR_S.UFRDY = '0';
    if HaveSecurityExt() then
        securefaultpri = ExceptionPriority(SecureFault, TRUE, FALSE);
        if SHCSR_S.SECUREFAULTENA == '1' && execPri > securefaultpri then
            FPCCR_S.SFRDY = '1';
        else
            FPCCR_S.SFRDY = '0';
    if CanPendMonitorOnEvent(isSecure, TRUE, TRUE) then
        FPCCR_S.MONRDY = '1';
    else
        FPCCR_S.MONRDY = '0';
    return;
```

### E2.1.373 ValidateAddress

```
// ValidateAddress()
// =================
```

```
3
4   (ExcInfo, AddressDescriptor) ValidateAddress(bits(32) address, AccType acctype,
5                                                boolean ispriv, boolean secure,
6                                                boolean iswrite, boolean aligned)
7       AddressDescriptor result;
8       Permissions       perms;
9       ns              = boolean UNKNOWN;
10      excInfo         = DefaultExcInfo();
11
12      // Security checking and MPU bank selection if Security Extensions are present.
13      if HaveSecurityExt() then
14          // Check SAU/IDAU for given address.
15          isInstrfetch = acctype == AccType_IFETCH;
16          sAttrib      = SecurityCheck(address, isInstrfetch, secure);
17          if isInstrfetch then
18              ns        = sAttrib.ns;
19              secureMpu = !sAttrib.ns;
20              // Override the privilege flag supplied with the a value based on the
21              // privilege associated with the current mode and the Security state
22              // of the MPU being queried. This can be different from value this
23              // function is called with, because CONTROL.nPRIV is banked between
24              // the Security states.
25              ispriv = CurrentModeIsPrivileged(secureMpu);
26          else
27              ns        = !secure || sAttrib.ns;
28              secureMpu = secure;
29      else
30          ns        = TRUE;
31          secureMpu = FALSE;
32
33      // Getting memory attribute information from MPU. Note that NS information
34      // in the memory attribute is set by  SAU/IDAU and is updated after getting
35      // attribute values from MPU.
36      (result.memattrs, perms) = MPUCheck(address, acctype, ispriv, secureMpu);
37      // Updating NS information got from SAU/IDAU in memory attributes
38      result.memattrs.NS       = ns;
39
40      // Generate UNALIGNED UsageFault exception if access to Device memory is unaligned.
41      if !aligned && result.memattrs.memtype == MemType_Device && perms.apValid == TRUE then
42          if acctype != AccType_DBG then
43              if secure then
44                  UFSR_S.UNALIGNED  = '1';
45              else
46                  UFSR_NS.UNALIGNED = '1';
47          excInfo = CreateException(UsageFault, FALSE, secure);
48
49      if excInfo.fault == NoFault && HaveSecurityExt() then
50          // Check if there is a SAU/IDAU violation and, if so, update the fault informations
51          case acctype of
52              when AccType_IFETCH
53                  if secure then
54                      if sAttrib.ns then
55                          // Invalid exit from the Secure state
56                          SFSR.INVTRAN = '1';
57                          excInfo      = CreateException(SecureFault);
58                  else
59                      if !sAttrib.ns && !sAttrib.nsc then
60                          // Invalid entry to the Secure state
61                          SFSR.INVEP   = '1';
62                          excInfo      = CreateException(SecureFault);
63              when AccType_VECTABLE
64                  if !secure && !sAttrib.ns then
65                      HFSR.VECTTBL = '1';
66                      // Vector fetche faults raise a HardFault directly, but because this
                            fault
67                      // is caused by an SAU/IDAU violation it always targets the secure state.
68                      excInfo      = CreateException(HardFault, TRUE, TRUE);
69              when AccType_DBG
70                  if !secure && !sAttrib.ns then
```

```
                    // DAP accesses result in a error being returned to the DAP without any
                    // syndrome being set.
                    excInfo        = CreateException(SecureFault);
            when AccType_NORMAL, AccType_MVE, AccType_ORDERED, AccType_STACK
                if !secure && !sAttrib.ns then
                    SFSR.AUVIOL    = '1';
                    SFSR.SFARVALID = '1';
                    SFAR           = address;
                    excInfo        = CreateException(SecureFault);
            when AccType_LAZYFP
                if !secure && !sAttrib.ns then
                    SFSR.LSPERR    = '1';
                    SFSR.SFARVALID = '1';
                    SFAR           = address;
                    excInfo        = CreateException(SecureFault);
            otherwise
                assert(FALSE);

    result.paddress         = address;
    result.accattrs.iswrite = iswrite;
    result.accattrs.ispriv  = ispriv;
    result.accattrs.acctype = acctype;

    if excInfo.fault == NoFault then
        excInfo = CheckPermission(perms, address, acctype, iswrite, ispriv, secureMpu);

    return (excInfo, result);
```

### E2.1.374 ValidateExceptionReturn

```
// ValidateExceptionReturn()
// =========================

(ExcInfo, EXC_RETURN_Type) ValidateExceptionReturn(EXC_RETURN_Type excReturn, integer
    returningExceptionNumber)
    boolean error         = FALSE;
    assert CurrentMode() == PEMode_Handler;
    if !IsOnes(excReturn<23:7>) || excReturn<1> != '0' then
        UNPREDICTABLE;
    if !HaveMveOrFPExt() && excReturn.FType == '0' then
        UNPREDICTABLE;
    if !HaveSecurityExt() && (excReturn.S    == '1' ||
                              excReturn.ES   == '1' ||
                              excReturn.DCRS == '0') then
        UNPREDICTABLE;

    // Security specific validation
    if HaveSecurityExt() then
        // If exception return is an invalid attempt to return from Non-secure
        // state with EXC_RETURN.ES set as '1', then a SecureFault is raised
        exceptionWasSecure = excReturn.ES == '1';
        if CurrentState == SecurityState_NonSecure && excReturn.ES == '1' then
            error              = TRUE;
            // excReturn.ES is used below to control which exception to
            // deactivate, and which CONTROL.SPSEL to update. Force it to the
            // correct value so the code below functions correctly even if the
            // Non-secure state returned an invalid excReturn value.
            // Similarly the exception to deactivate below is actually Non-secure
            excReturn.ES       = '0';
            exceptionWasSecure = FALSE;

        // Check DCRS bit not used in for Non-secure exceptions
        if !exceptionWasSecure && excReturn.DCRS == '0' then
            error = TRUE;

        if error then
            SFSR.INVER      = '1';
            exceptionNumber = SecureFault;
```

```
    else
        exceptionWasSecure = FALSE;

    // check returning from an inactive handler
    if !error then
        if !IsActiveForState(returningExceptionNumber, exceptionWasSecure) then
            error = TRUE;
            if HaveMainExt() then
                UFSR.INVPC      = '1';
                exceptionNumber = UsageFault;
            else
                exceptionNumber = HardFault;

    if error then
        DeActivate(returningExceptionNumber, exceptionWasSecure);
        if HaveSecurityExt() && exceptionWasSecure then
            CONTROL_S.SPSEL  = excReturn.SPSEL;
        else
            CONTROL_NS.SPSEL = excReturn.SPSEL;
        // Escalates to HardFault if requested fault is disabled, or has
        // insufficient priority, or if Main Extension is not implemented
        excInfo = CreateException(exceptionNumber, FALSE, IsSecure());
    else
        excInfo = DefaultExcInfo();
    return (excInfo, excReturn);
```

## E2.1.375 Vector

```
// Vector[]
// ========

(ExcInfo, bits(32)) Vector[integer exceptionNumber, boolean isSecure]
    // Calculate the address of the entry in the vector table
    vtor = if isSecure then VTOR_S else VTOR_NS;
    addr = (vtor.TBLOFF:'0000000') + 4 * exceptionNumber;
    // Fetch the vector with the correct privilege and security
    (exc, vector) = MemA_with_priv_security(addr,4,AccType_VECTABLE,TRUE,isSecure,TRUE);
    // Faults that prevent the vector being fetched are terminal and prevent
    // the exception being entered.
    if exc.fault != NoFault then
        exc.isTerminal = TRUE;
    return (exc, vector);
```

## E2.1.376 VectorCatchDebug

```
// VectorCatchDebug()
// ==================

VectorCatchDebug()
    vectorEvt = FALSE;

    if CanHaltOnEvent(IsSecure()) then
        case UInt(IPSR.Exception) of
            when HardFault
                vectorEvt = ((DEMCR.VC_HARDERR == '1' && (HFSR.FORCED      == '1'   ||
                                                          HFSR.DEBUGEVT    == '1')) ||
                             (DEMCR.VC_INTERR  == '1' && (HFSR.VECTTBL     == '1')));

            when MemManage
                vectorEvt = ((DEMCR.VC_MMERR   == '1' && (MMFSR.IACCVIOL   == '1'   ||
                                                          MMFSR.DACCVIOL   == '1')) ||
                             (DEMCR.VC_INTERR  == '1' && (MMFSR.MSTKERR    == '1'   ||
                                                          MMFSR.MUNSTKERR  == '1'   ||
                                                          MMFSR.MLSPERR    == '1')));

            when BusFault
                vectorEvt = ((DEMCR.VC_BUSERR  == '1' && (BFSR.IBUSERR     == '1'   ||
```

```
                                                    BFSR.PRECISERR   == '1'   ||
                                                    BFSR.IMPRECISERR == '1')) ||
                              (DEMCR.VC_INTERR  == '1' && (BFSR.STKERR      == '1'   ||
                                                    BFSR.UNSTKERR    == '1'   ||
                                                    BFSR.LSPERR      == '1')));

            when UsageFault
                vectorEvt = ((DEMCR.VC_STATERR == '1' && (UFSR.UNDEFINSTR  == '1'   ||
                                                    UFSR.INVPC       == '1'   ||
                                                    UFSR.INVSTATE    == '1')) ||
                              (DEMCR.VC_CHKERR  == '1' && (UFSR.UNALIGNED   == '1'   ||
                                                    UFSR.DIVBYZERO   == '1')) ||
                              (DEMCR.VC_INTERR  == '1' && (UFSR.STKOF       == '1')) ||
                              (DEMCR.VC_NOCPERR == '1' && (UFSR.NOCP        == '1')));

            when SecureFault
                vectorEvt =  ((DEMCR.VC_SFERR  == '1' && (SFSR.INVEP       == '1'   ||
                                                    SFSR.INVIS       == '1'   ||
                                                    SFSR.INVER       == '1'   ||
                                                    SFSR.AUVIOL      == '1'   ||
                                                    SFSR.INVTRAN     == '1'   ||
                                                    SFSR.LSPERR      == '1'   ||
                                                    SFSR.LSERR       == '1')));

            otherwise
                // No other exceptions trigger vector catch

        if vectorEvt then
           DHCSR.C_HALT = '1';
           DFSR.VCATCH  = '1';
```

## E2.1.377 VFPExcBarrier

```
// VFPExcBarrier
// =============
// Ensures that all floating-point exception processing has completed

VFPExcBarrier();
```

## E2.1.378 VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N IN {16,32,64};
    integer E = if N == 16 then 5 elsif N == 32 then 8 else 11;
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3);
    frac = imm8<5:0>:Zeros(F-4);
    return sign : exp : frac;
```

## E2.1.379 VFPNegMul

```
// Different types of floating-point multiply and negate operations

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

## E2.1.380 VFPSmallRegisterBank

```
// VFPSmallRegisterBank()
// ======================
// Returns TRUE because the Floating Point implementation only provides access to
```

```
// 16 double-precision registers

boolean VFPSmallRegisterBank()
    return TRUE;
```

### E2.1.381 ViolatesSPLim

```
// ViolatesSPLim()
// ===============

boolean ViolatesSPLim(RNames spreg, bits(32) value)
    isSecure = ((spreg == RNamesSP_Main_Secure) || (spreg == RNamesSP_Process_Secure));

    // Check CCR.STKOFHFNMIGN to determine if the limit should actually be
    // applied. When checking if CCR.STKOFHFNMIGN should apply the requested
    // execution priority is considered, and AIRCR.PRIS is ignored.
    assert (!isSecure || HaveSecurityExt());
    if HaveMainExt() && IsReqExcPriNeg(isSecure) then
        ignLimit   = if isSecure then CCR_S.STKOFHFNMIGN else CCR_NS.STKOFHFNMIGN;
        applylimit = (ignLimit == '0');
    else
        applylimit = TRUE;

    return applylimit && (UInt(value) < UInt(LookUpSPLim(spreg)));
```

### E2.1.382 VPTActive

```
// VPTActive()
// ===========

boolean VPTActive()
    return VPTActive(_BeatID);

boolean VPTActive(integer beat)
    return Elem[VPR<23:16>, beat DIV 2, 4] != Zeros(4);
```

### E2.1.383 VPTAdvance

```
// VPTAdvance()
// ============

VPTAdvance(integer beat)
    maskID   = beat DIV 2;
    vptState = Elem[VPR<23:16>, maskID, 4];
    if vptState == '1000' then
        vptState = Zeros(4);
    elsif vptState != '0000' then
        (vptState, inv) = LSL_C(vptState, 1);
        // Invert the predicate flags for this beat if the bit shifted out of
        // the VPT state was 1.
        if inv == '1' then
            Elem[VPR.P0, beat, 4] = NOT Elem[VPR.P0, beat, 4];
    // Since the mask fields are grouped in pairs only update the mask on every odd numbered
        beat.
    if beat<0> == '1' then
        Elem[VPR<23:16>, maskID, 4] = vptState;
```

### E2.1.384 WaitForEvent

```
// WaitForEvent
// ============
// Optionally suspends execution until a WFE wakeup event or reset occurs,
// or until some earlier time if the implementation chooses

WaitForEvent();
```

### E2.1.385 WaitForInterrupt

```
// WaitForInterrupt
// ================
// Optionally suspends execution until a WFI wakeup event or reset occurs, or
// until some earlier time if the implementation chooses

WaitForInterrupt();
```

### E2.1.386 zeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

### E2.1.387 zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

bits(N) Zeros()
    return Zeros(N);
```

# Part F
## Debug Packet Protocols

# Chapter F1
# ITM and DWT Packet Protocol Specification

This chapter describes the protocol for packets that send the data generated by the ITM and DWT to an external debugger. It contains the following sections:

- About the ITM and DWT packets.
- Alphabetical list of DWT and ITM packets.

# F1.1 About the ITM and DWT packets

The following sections give an overview of the ITM and DWT packets and how the TPIU transmits them:

- Uses of ITM and DWT packets
- ITM and DWT protocol packet headers
- Packet transmission by the trace sink

**Note**

This chapter describes packet transmission by a trace sink such as a TPIU. The ITM can send packets to any suitable trace sink. Regardless of the actual trace sink used, the ITM formats the packets as described in this chapter.

## F1.1.1 Uses of ITM and DWT packets

The ITM sends a packet to the trace sink when:

- Software writes to a stimulus register. This generates a Instrumentation packet.
- The hardware generates a Protocol packet. Protocol packets include timestamps and synchronization packets.
- It receives a packet from the DWT, for forwarding to the trace sink.

The DWT sends a packet to the ITM for forwarding to the trace sink when:

- A DWT comparator matches and generates one or more Data Trace packets.
- It samples the PC.
- One of the performance profile counters wraps.

This chapter describes the packet protocol used.

## F1.1.2 ITM and DWT protocol packet headers

| [7] | [6] | [5] | [4] | [3] | [2] | [1] | [0] | Description |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | Synchronization packet |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | Overflow packet |
| 0 | $\neq$0b000 && $\neq$ 0b111 | | | 0 | 0 | 0 | 0 | Local Timestamp 2 packet |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Global Timestamp 1 packet |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | Global Timestamp 2 packet |
| 1 | 1 | x | x | 0 | 0 | 0 | 0 | Local Timestamp 1 packet |
| x | x | x | x | 1 | x | 0 | 0 | Extension Packet |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | Event Counter Packet |
| 0 | 1 | x | x | 0 | 1 | $\neq$0b00 | | Data Trace PC Value packet |
| 0 | 1 | x | x | 1 | 1 | $\neq$0b00 | | Data Trace Data Address packet |
| 1 | 0 | x | x | x | 1 | $\neq$0b00 | | Data Trace Data Value packet |
| x | x | x | x | x | 0 | $\neq$0b00 | | Instrumentation packet |
| 0 | 0 | 0 | 1 | 0 | 1 | x | 1 | Periodic PC Sample packet |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | PMU Overflow packet |

### F1.1.3 Packet transmission by the trace sink

The trace sink either:

- Forms the packets into frames, as required by the *Arm®CoreSight™ Architecture Specification.*
- Transmits the packets over a serial port.

For each packet, the trace sink transmits:

- The header byte first, followed by any payload bytes.
- Each byte of the packet *least significant bit* (LSB) first.

Figures in this chapter show each packet as a sequence of bytes, with the LSB of each byte to the right and the *most significant bit* (MSB) to the left. Convention for packet descriptions shows this convention, and how it relates to data transmission for a packet with a header byte and two payload bytes.

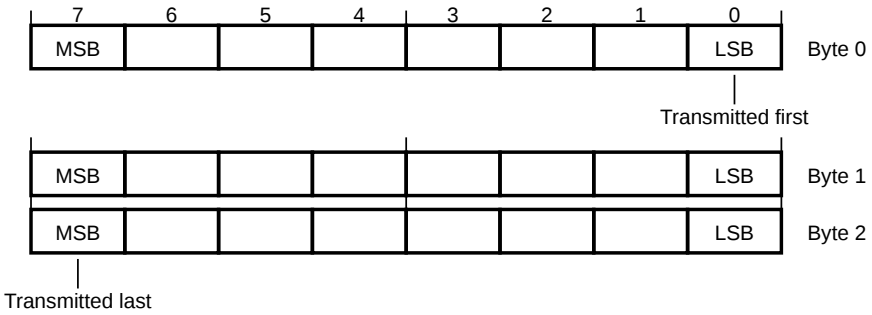



**Figure F1.1: Convention for packet descriptions**

In some sections, the figures are split into separate figures for the header byte and payload bytes. For instance, where the number of payload bytes varies according to a field in the header.

The ITM merges the packets from the ITM and DWT with the Local and Global timestamp, Synchronization, and other Protocol packets, and forwards them to the trace sink as a single data stream. The trace sink then merges this data stream with the data from the ETM, if implemented.

# F1.2 Alphabetical list of DWT and ITM packets

## F1.2.1 Data Trace Data Address packet

The Data Trace Data Address packet characteristics are:

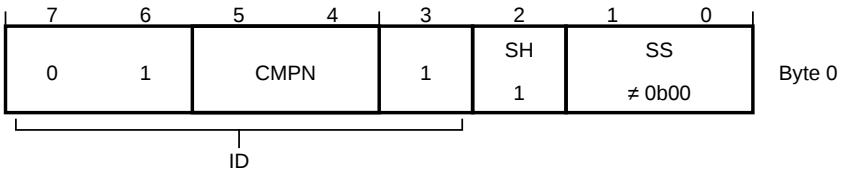
**Purpose** Indicates a DWT comparator generated a match, and the address that matched. Data Address packets are only generated for Data Address range comparator pairs. The address might be compressed. However, it is not required that Short and Medium packets are generated when the address bits match.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

### F1.2.1.1 Data Trace Data Address packet header

The Data Trace Data Address packet header bit assignments are:




**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**`0b01xx1`** Data Trace Data Address packet.

This field reads as `0b01xx1`.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match. Data Trace Data Address packets can be compressed relative to the value in DWT_COMP<*CMPN*>. The number of traced bits is indicated by the SS field. The remainder of the address bits comes from DWT_COMP<*CMPN*>. Either comparator in a Data Address range comparator pair can be used.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**`1`** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**`0b01`** Short Data Address packet.

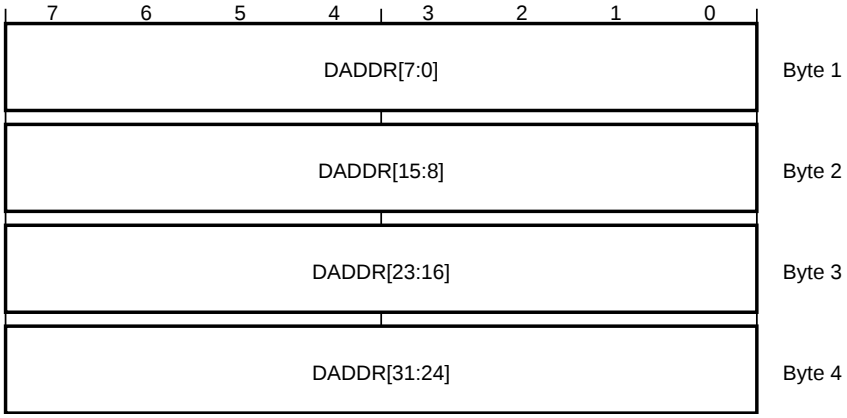
**`0b10`** Medium Data Address packet.

**`0b11`** Long Data Address packet.

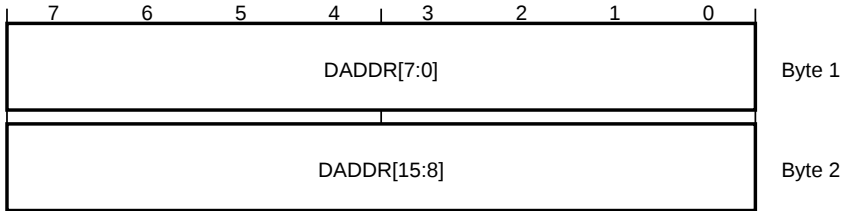
The value `0b00` encodes a Protocol packet.

#### F1.2.1.2 Data Trace Data Address packet payload

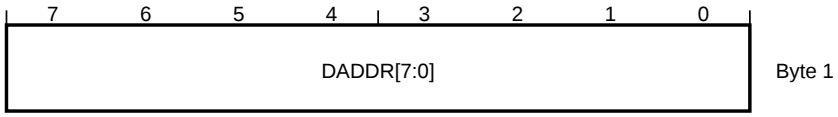
When Long Data Address packet, SS == `0b11`, the Data Trace Data Address packet payload bit assignments are:




When Medium Data Address packet, SS == `0b10`, the Data Trace Data Address packet payload bit assignments are:




When Short Data Address packet, SS == `0b01`, the Data Trace Data Address packet payload bit assignments are:




**DADDR[31:0], bytes <4:1>, when Long Data Address packet, SS == `0b11`** Data address.

**DADDR[15:0], bytes <2:1>, when Medium Data Address packet, SS == `0b10`** Data address. DADDR[31:16] == DWT_COMP<*CMPN*>[31:16].

**DADDR[7:0], byte <1>, when Short Data Address packet, SS == `0b01`** Data address. DADDR[31:8] == DWT_COMP<*CMPN*>[31:8].

### F1.2.2 Data Trace Data Value packet

The Data Trace Data Value packet characteristics are:

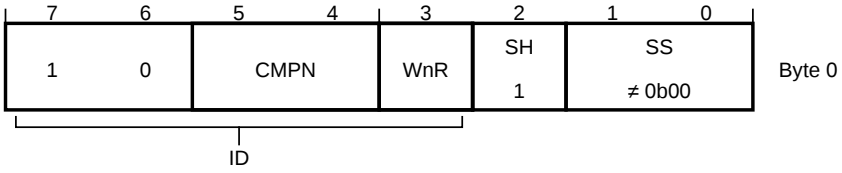
**Purpose** Indicates a DWT comparator generated a match, and the value that matched.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

### F1.2.2.1 Data Trace Data Value packet header

The Data Trace Data Value packet header bit assignments are:




**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**`0b10xxx`** Data Trace Data Value packet.

This field reads as `0b10xxx`.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match.

**WnR, byte 0 bit [3]** Write-not-read. The defined values of this bit are:

**`0`** Read.

**`1`** Write.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**`1`** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**`0b01`** Byte Data Value packet.

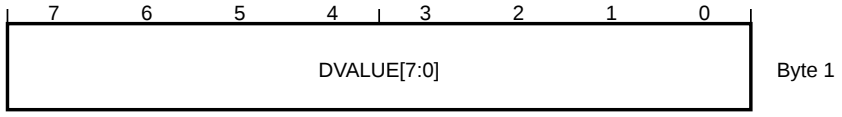
**`0b10`** Halfword Data Value packet.

**`0b11`** Word Data Value packet.

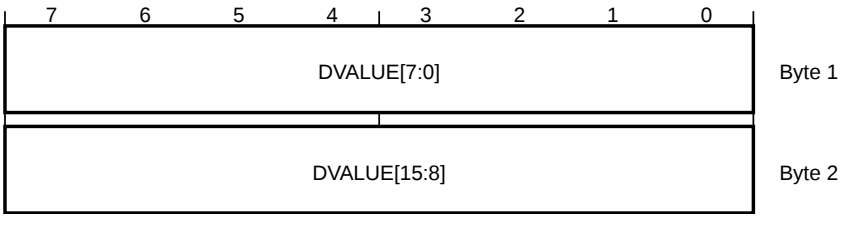
The value `0b00` encodes a Protocol packet.

### F1.2.2.2 Data Trace Data Value packet payload

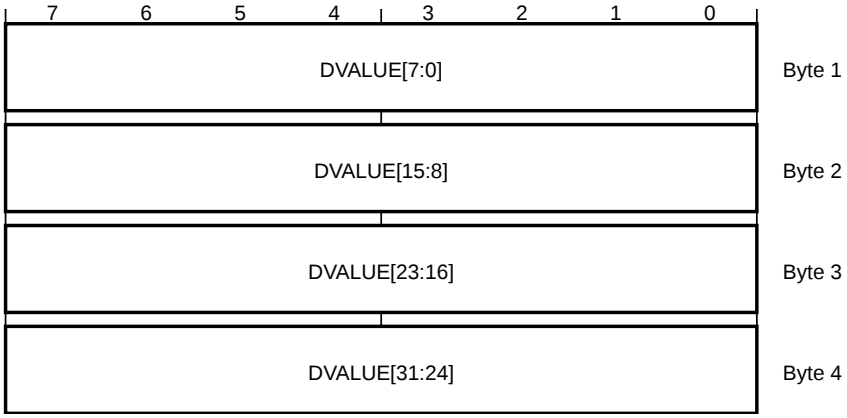
When Byte Data Value packet, SS == `0b01`, the Data Trace Data Value packet payload bit assignments are:




When Halfword Data Value packet, SS == `0b10`, the Data Trace Data Value packet payload bit assignments are:



When Word Data Value packet, SS == `0b11`, the Data Trace Data Value packet payload bit assignments are:




**DVALUE[31:0], bytes <4:1>, when Word Data Value packet, SS == `0b11`** Word data value.

**DVALUE[15:0], byte 1 bits [15:0], when Halfword Data Value packet, SS == `0b10`** Halfword data value.

**DVALUE[7:0], byte <1>, when Byte Data Value packet, SS == `0b01`** Byte data value.

### F1.2.3 Data Trace Match packet

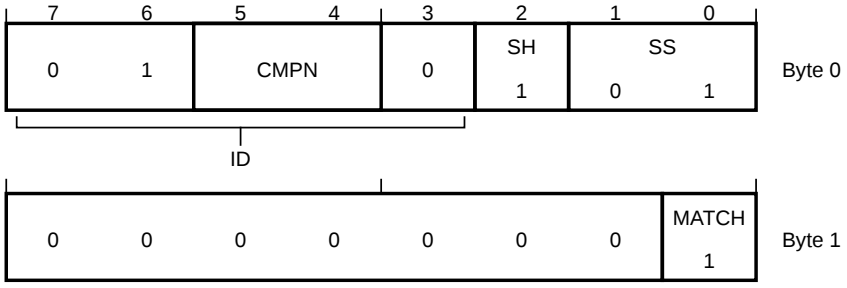
The Data Trace Match packet characteristics are:

**Purpose** Indicates a DWT comparator generated a match.

**Attributes** 16-bit Hardware source packet.

#### Field descriptions

The Data Trace Match packet bit assignments are:




**Byte 1 bits [7:1]** This field reads as `0b0000000`.

**MATCH, byte 1 bit [0]** Data Trace Match packet. Discriminates between the Data Trace PC Value packet and the Data Trace Match packet. The defined values of this bit are:

**`1`** Data Trace Match packet.

This bit reads as one.

**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b01xx0** Data Trace PC Value packet or Data Trace Match packet.

Bit [0] of byte 1 discriminates between the Data Trace PC Value packet and the Data Trace Match packet.

This field reads as 0b01xx0.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Source packet, 1-byte payload, 2-byte packet.

The value 0b00 encodes a Protocol packet. All other values are reserved.

This field reads as 0b01.

### F1.2.4 Data Trace PC Value packet

The Data Trace PC Value packet characteristics are:

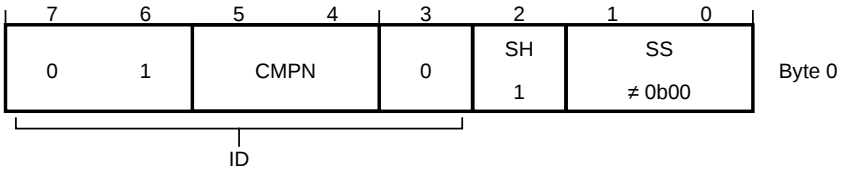
**Purpose** Indicates a DWT comparator generated a match, and the address of the instruction that matched. The address might be compressed. However, it is not required that Short and Medium packets are generated when the address bits match.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

#### F1.2.4.1 Data Trace PC Value packet header

The Data Trace PC Value packet header bit assignments are:




**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**0b01xx0** Data Trace PC Value packet or Data Trace Match packet.

Bit [0] of byte 1 discriminates between the Data Trace PC Value packet and the Data Trace Match packet.

This field reads as 0b01xx0.

**CMPN, byte 0 bits [5:4]** DWT comparator index. Defines which comparator generated a match. Data Trace PC Value packets can be compressed relative to the value in DWT_COMP<*CMPN*>. The number of traced bits is indicated by the SS field. The remainder of the address bits comes from DWT_COMP<*CMPN*>. Either comparator in an Instruction Address range comparator pair can be used.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Short PC Value packet.

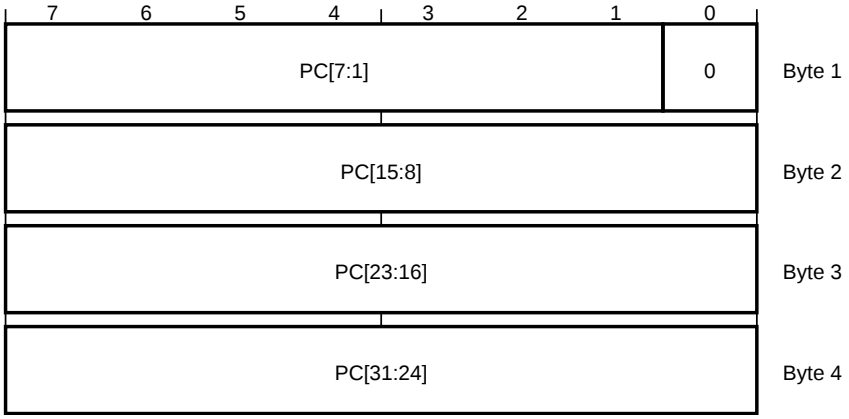
**0b10** Medium PC Value packet.

**0b11** Long PC Value packet.

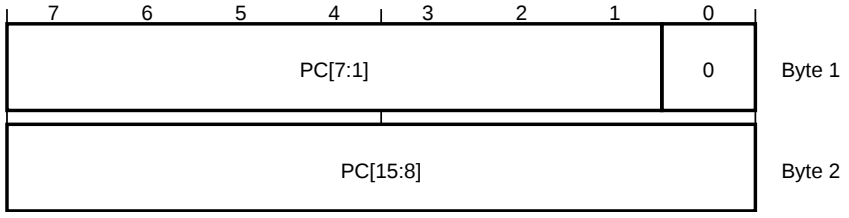
The value 0b00 encodes a Protocol packet.

### F1.2.4.2 Data Trace PC Value packet payload

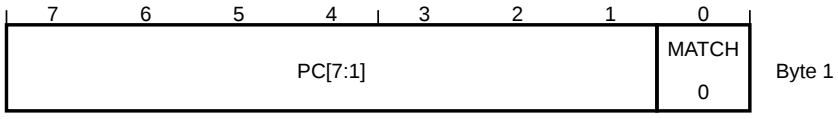
When Long PC Value packet, SS == 0b11, the Data Trace PC Value packet payload bit assignments are:




When Medium PC Value packet, SS == 0b10, the Data Trace PC Value packet payload bit assignments are:




When Short PC Value packet, SS == 0b01, the Data Trace PC Value packet payload bit assignments are:




**PC[31:1], bytes <4:2>, byte 1 bits [7:1], when Long PC Value packet, SS == 0b11** Instruction address.

**PC[15:1], byte <2>, byte 1 bits [7:1], when Medium PC Value packet, SS == 0b10** Instruction address. PC[31:16] == DWT_COMP<*CMPN*>[31:16].

**PC[7:1], byte 1 bits [7:1], when Short PC Value packet, SS == 0b01** Instruction address. PC[31:8] == DWT_COMP<*CMPN*>[31:8].

**MATCH, byte 1 bit [0]** Data Trace Match packet. Discriminates between the Data Trace PC Value packet and the Data Trace Match packet. The defined values of this bit are:

`0` Data Trace PC Value packet.

This bit reads as zero.

### F1.2.5 Event Counter packet

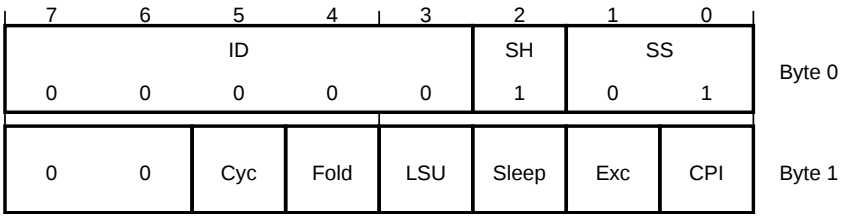
The Event Counter packet characteristics are:

**Purpose** Indicates one or more DWT counters wraps through zero.

**Attributes** 16-bit Hardware source packet.

#### Field descriptions

The Event Counter packet bit assignments are:




**Byte 1 bits [7:6]** This field reads-as-zero.

**Cyc, byte 1 bit [5]** POSTCNT timer decremented to zero. See DWT_CTRL for more information on the POSTCNT timer.

**Fold, byte 1 bit [4]** DWT_FOLDCNT counter wrapped from `0xFF` to zero.

**LSU, byte 1 bit [3]** DWT_LSUNCT counter wrapped from `0xFF` to zero.

**Sleep, byte 1 bit [2]** DWT_SLEEPCNT counter wrapped from `0xFF` to zero.

**Exc, byte 1 bit [1]** DWT_EXCCNT counter wrapped from `0xFF` to zero.

**CPI, byte 1 bit [0]** DWT_CPICNT counter wrapped from `0xFF` to zero.

**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

`0b00000` Event Counter packet.

This field reads as `0b00000`.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

`1` Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

`0b01` Source packet, 1-byte payload, 2-byte packet.

The value `0b00` encodes a Protocol packet. All other values are reserved.

This field reads as `0b01`.

### F1.2.6 Exception Trace packet

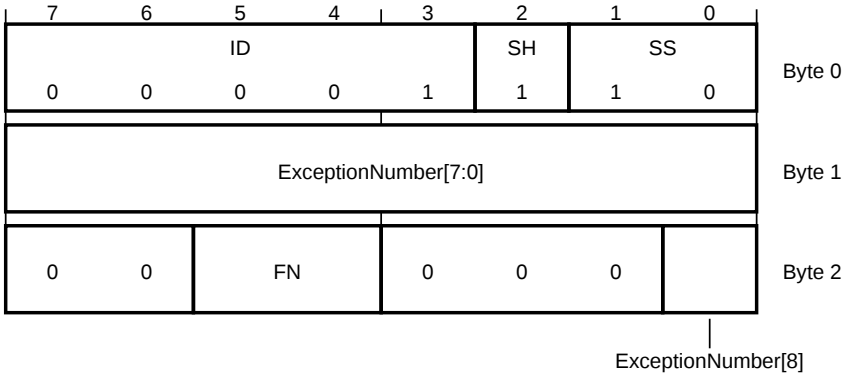
The Exception Trace packet characteristics are:

**Purpose** Indicates the PE has entered, exited or returned to an exception.

**Attributes** 24-bit Hardware source packet.

#### Field descriptions

The Exception Trace packet bit assignments are:




**Byte 2 bits [7:6,3:1]** This field reads-as-zero.

**FN, byte 2 bits [5:4]** Function. The defined values of this field are:

**`0b01`** Entered exception indicated by ExceptionNumber.

**`0b10`** Exited exception indicated by ExceptionNumber.

**`0b11`** Returned to exception indicated by ExceptionNumber.

All other values are reserved.

**ExceptionNumber, byte 2 bit [0], byte <1>** The exception number.

**ID, byte 0 bits [7:3]** Hardware Source packet type. Bits [7:3] discriminate between Hardware Source packet types. The defined values of this field are:

**`0b00001`** Exception Trace packet.

This field reads as `0b00001`.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**`1`** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**`0b10`** Source packet, 2-byte payload, 3-byte packet.

The value `0b00` encodes a Protocol packet. All other values are reserved.

This field reads as `0b10`.

### F1.2.7 Extension packet

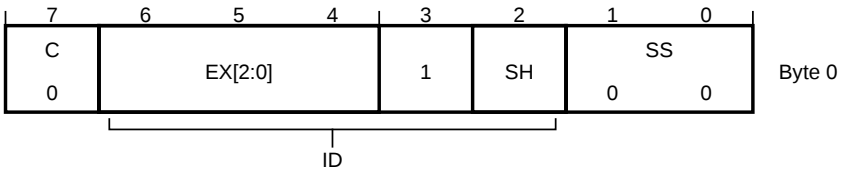
The Extension packet characteristics are:

**Purpose** An Extension packet provides additional information about the identified source. The amount of information required determines the number of payload bytes, 0-4. The architecture only defines one use of the Extension packet, to provide a Stimulus port page number. For this use, SH == 0, and a single byte Extension packet is emitted.

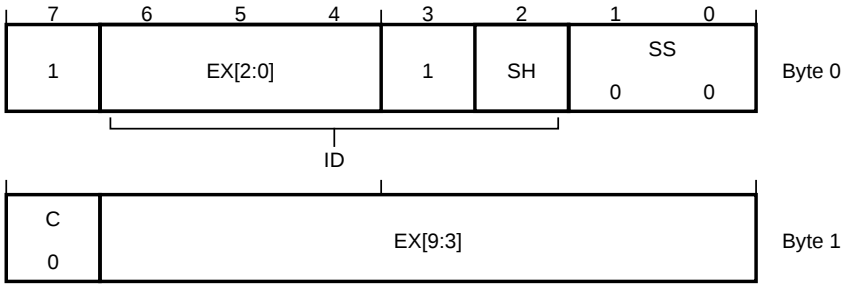
**Attributes** 8, 16, 24, 32, or 40-bit Protocol packet.

#### Field descriptions

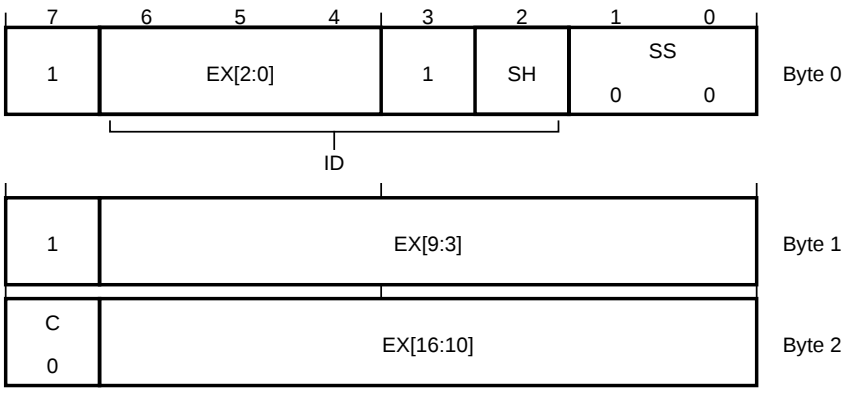
When 1-byte packet, the Extension packet bit assignments are:



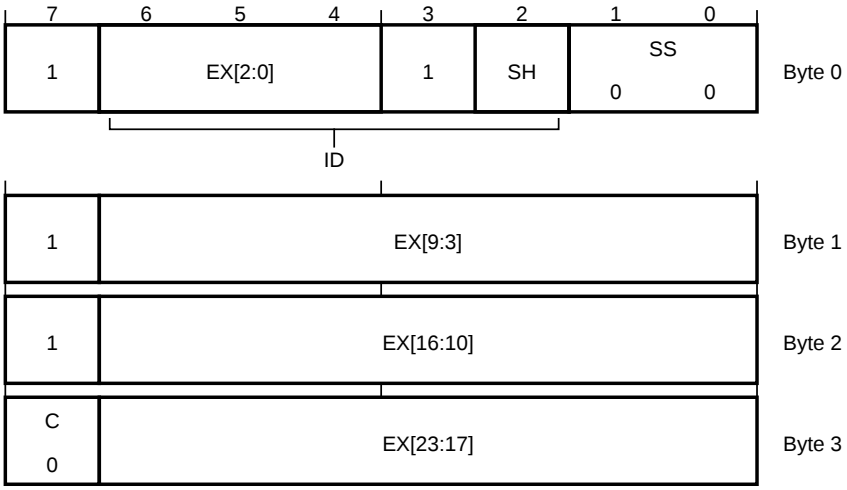

When 2-byte packet, the Extension packet bit assignments are:



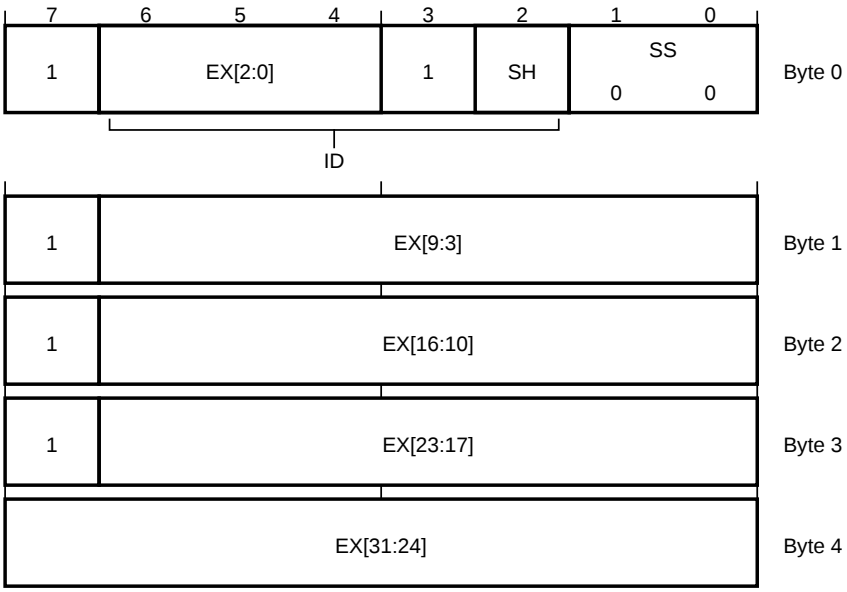

When 3-byte packet, the Extension packet bit assignments are:




When 4-byte packet, the Extension packet bit assignments are:




When 5-byte packet, the Extension packet bit assignments are:




**EX, byte <4>, byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], byte 0 bits [6:4]** Extension information. If SH == 1, then EX defines PAGE, the Stimulus port page number.

This is a 32-bit field. If the Extension packet is shorter than 5 bytes, the most significant bits are zero.

**C, byte 3 bit [7], byte 2 bit [7], byte 1 bit [7], byte 0 bit [7]** Continuation bit. The defined values of this field are:

`0` Last byte of the packet.

`1` Another byte follows.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**`0bxxx1x`** Extension packet.

This field reads as `0bxxx1x`.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**0** Extension packet for Instrumentation packet.

**1** Extension packet for Hardware source packet.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

## F1.2.8 Global Timestamp 1 packet

The Global Timestamp 1 packet characteristics are:

**Purpose** Contains the least significant bits of the global timestamp value. The ITM might compress this value if it is not generating a full timestamp by omitting significant bits if they are unchanged from the previous timestamp value.

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 8, 16, 24, or 32-bit payload.

### F1.2.8.1 Global Timestamp 1 packet header

The Global Timestamp 1 packet header bit assignments are:


**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b00101** Global Timestamp 1 packet.

This field reads as 0b00101.

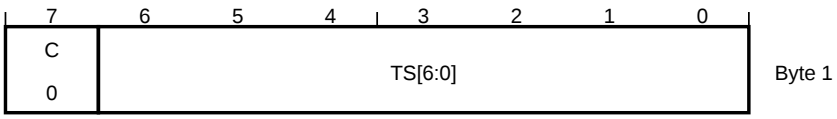
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

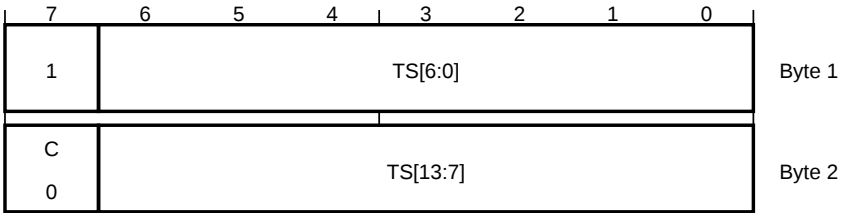
Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.8.2 Global Timestamp 1 packet payload

When 7-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:

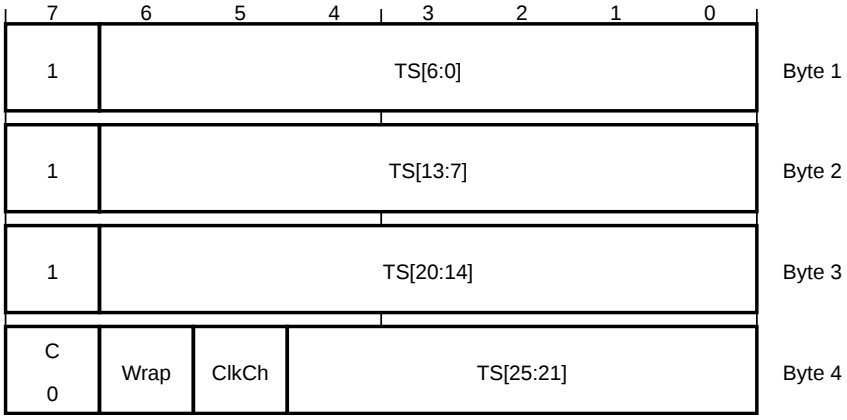



When 14-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:




When 21-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:


When 26-bit or full timestamp, the Global Timestamp 1 packet payload bit assignments are:




**C, byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7]** Continuation bit. The defined values of this field are:

**0** Last byte of the packet.

**1** Another byte follows.

**Wrap, byte 4 bit [6], when 26-bit or full timestamp** Wrapped. The defined values of this bit are:

**0** The value of global timestamp bits TS[47:26] or TS[63:26] have not changed since the last Global Timestamp 2 packet output by the ITM.

**1** The value of global timestamp bits TS[47:26] or TS[63:26] have changed since the last Global Timestamp 2 packet output by the ITM.

**ClkCh, byte 4 bit [5], when 26-bit or full timestamp** Clock change. The defined values of this bit are:

**0** The system has not asserted the clock change input to the processor since the last time the ITM generated a Global Timestamp packet.

**1** The system has asserted the clock change input to the processor since the last time the ITM generated a Global Timestamp packet.

**Note**

When the clock change input to the processor is asserted, the ITM must output a full 48-bit or 64-bit global timestamp value.

**TS[25:0], byte 4 bits [4:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0]** Global Timestamp. The timestamp is 64 or 48 bits. If the Global Timestamp 1 packet is shorter than 5 bytes, the most-significant bits of the timestamp have not changed since the last Global Timestamp 1 packet output by the ITM. If the Global Timestamp 1 packet is 5 bytes, the Wrap bit defines whether most-significant bits have unchanged since the last Global Timestamp 2 packet output by the ITM.

## F1.2.9 Global Timestamp 2 packet

The Global Timestamp 2 packet characteristics are:

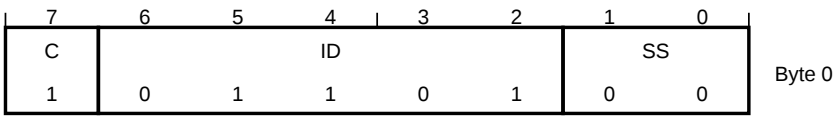
**Purpose** Provides the most significant bits of a full 48 or 64-bit timestamp.

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 32 or 48-bit payload.

### F1.2.9.1 Global Timestamp 2 packet header

The Global Timestamp 2 packet header bit assignments are:




**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**`0b01101`** Global Timestamp 2 packet.

This field reads as `0b01101`.

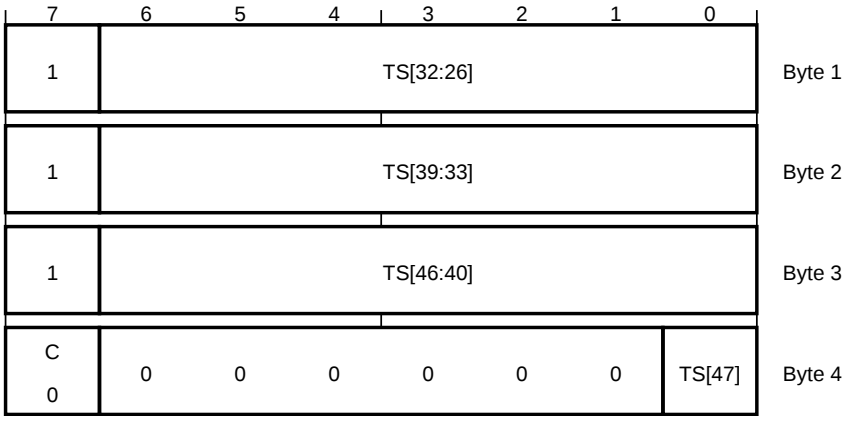
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**`0b00`** Protocol packet.

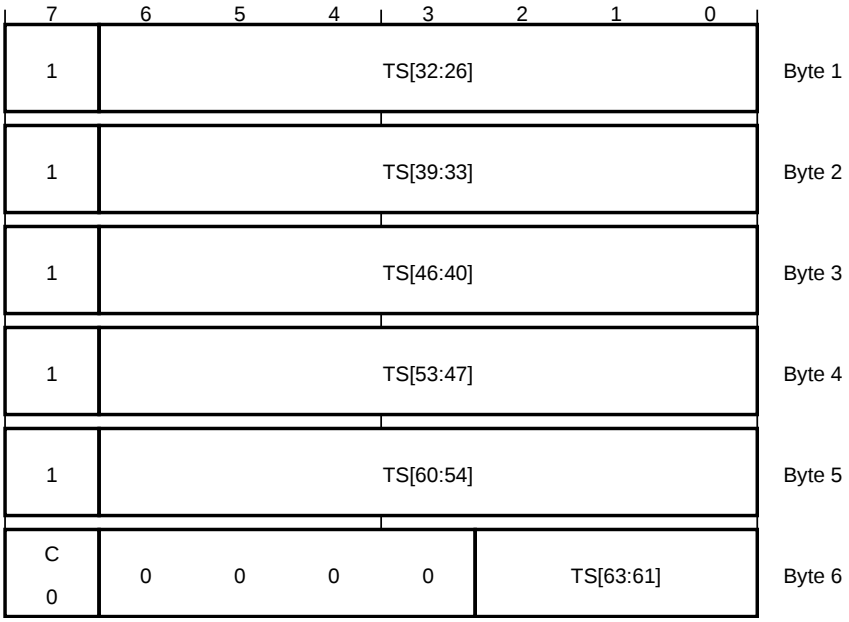
Other values encode different sizes of Hardware and Software source packets. This field reads as `0b00`.

### F1.2.9.2 Global Timestamp 2 packet payload

When 48-bit Global Timestamp 2 packet, the Global Timestamp 2 packet payload bit assignments are:




When 64-bit Global Timestamp 2 packet, the Global Timestamp 2 packet payload bit assignments are:




**C, byte 6 bit [7], byte 5 bit [7], byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7] Continuation bit.**
The defined values of this field are:

`0` Last byte of the packet.

`1` Another byte follows.

**Byte 6 bits [6:3], when 64-bit Global Timestamp 2 packet** This field reads-as-zero.

**Byte 4 bits [6:1], when 48-bit Global Timestamp 2 packet** This field reads-as-zero.

**TS[47:26] , byte 4 bit [0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], when 48-bit Global Timestamp 2 packet**

Most significant bits of the Global Timestamp.

**TS[63:26] , byte 6 bits [2:0], byte 5 bits [6:0], byte 4 bits [6:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0], when 64-bit Global Timestamp 2 packet**

Most significant bits of the Global Timestamp.

### F1.2.10 Instrumentation packet

The Instrumentation packet characteristics are:

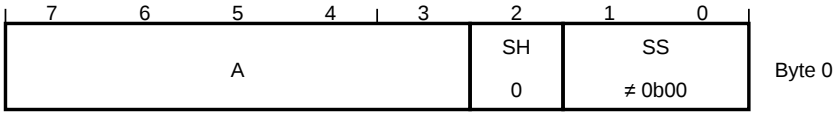
**Purpose** A software write to an ITM stimulus port generates an Instrumentation packet.

**Attributes** Multi-part Software source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

#### F1.2.10.1 Instrumentation packet header

The Instrumentation packet header bit assignments are:




**A, byte 0 bits [7:3]** Port number, 0-31.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**0** Instrumentation packet (Software source).

This bit reads as zero.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Byte Instrumentation packet.

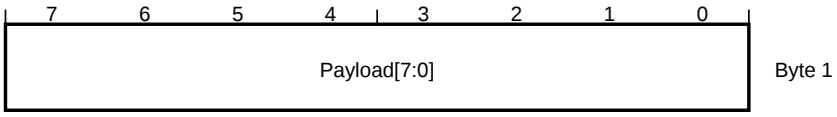
**0b10** Halfword Instrumentation packet.

**0b11** Word Instrumentation packet.

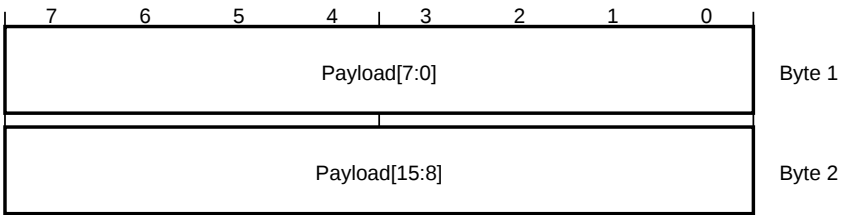
The value `0b00` encodes a Protocol packet.

#### F1.2.10.2 Instrumentation packet payload

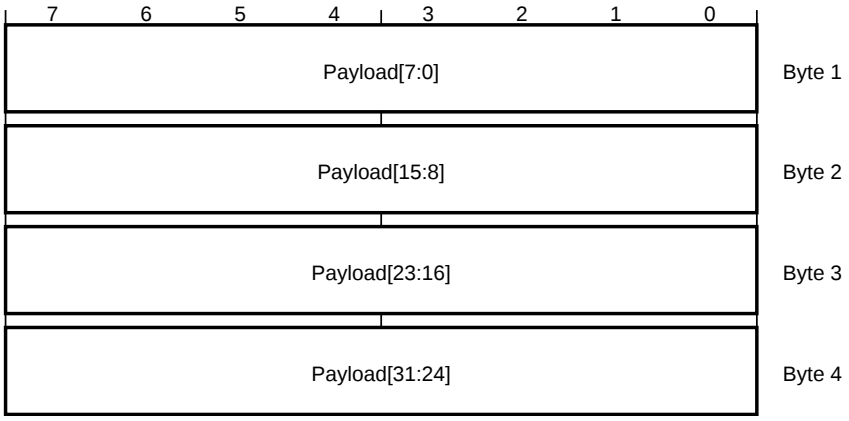
When Byte Instrumentation packet, SS == `0b01`, the Instrumentation packet payload bit assignments are:




When Halfword Instrumentation packet, SS == `0b10`, the Instrumentation packet payload bit assignments are:




When Word Instrumentation packet, SS == `0b11`, the Instrumentation packet payload bit assignments are:




**Payload[31:0], bytes <4:1>, when Word Instrumentation packet, SS == `0b11`** Payload value.

**Payload[15:0], byte 1 bits [15:0], when Halfword Instrumentation packet, SS == `0b10`** Payload value.

**Payload[7:0], byte <1>, when Byte Instrumentation packet, SS == `0b01`** Payload value.

### F1.2.11 Local Timestamp 1 packet

The Local Timestamp 1 packet characteristics are:

**Purpose** A Local Timestamp 1 packet encodes timestamp information, for generic control and synchronization, based on a timestamp counter in the ITM. To reduce the trace bandwidth:

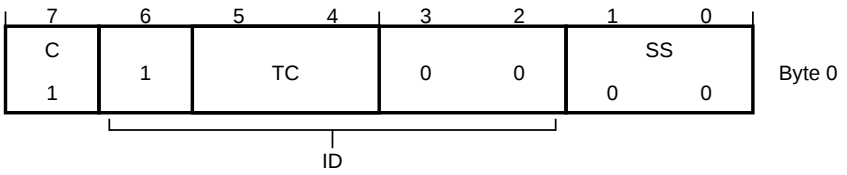
- The local timestamping scheme uses delta timestamps. Whenever the ITM outputs a Local timestamp packet, it clears its timestamp counter to zero, meaning each local timestamp value gives the interval since the generation of the previous Local timestamp packet.
- The Local Timestamp 1 packet length, 1-5 bytes, depends on the timestamp value.
- If the ITM outputs the local timestamp synchronously to the corresponding ITM or DWT data, and the timestamp value is in the range 1-6, the ITM uses the Local Timestamp 2 packet.

**Attributes** Multi-part Protocol packet comprising:

- 8-bit header.
- 8, 16, 24, or 32-bit payload.

#### F1.2.11.1 Local Timestamp 1 packet header

The Local Timestamp 1 packet header bit assignments are:




**C, byte 0 bit [7]** Continuation bit. This bit reads as one.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**`0b1xx00`** Local Timestamp 1 packet.

This field reads as `0b1xx00`.

**TC, byte 0 bits [5:4]** Indicates the relationship between the generation of the Local timestamp packet and the corresponding ITM or DWT data packet. The defined values of this field are:

**`0b00`** The local timestamp value is synchronous to the corresponding ITM or DWT data. The value in the TS field is the timestamp counter value when the ITM or DWT packet is generated.

**`0b01`** The local timestamp value is delayed relative to the ITM or DWT data. The value in the TS field is the timestamp counter value when the Local timestamp packet is generated.

**Note**

The local timestamp value corresponding to the previous ITM or DWT packet is *unknown*, but must be between the previous and current local timestamp values.

**`0b10`** Output of the ITM or DWT packet corresponding to this Local timestamp packet is delayed relative to the associated event. The value in the TS field is the timestamp counter value when the ITM or DWT packets is generated.

This encoding indicates that the ITM or DWT packet was delayed relative to other trace output packets.

**`0b11`** Output of the ITM or DWT packet corresponding to this Local timestamp packet is delayed relative to the associated event, and this Local timestamp packet is delayed relative to the ITM or DWT data. This is a combination of the conditions indicated by values `0b01` and `0b10`.

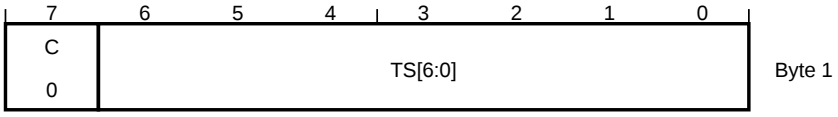
**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**`0b00`** Protocol packet.

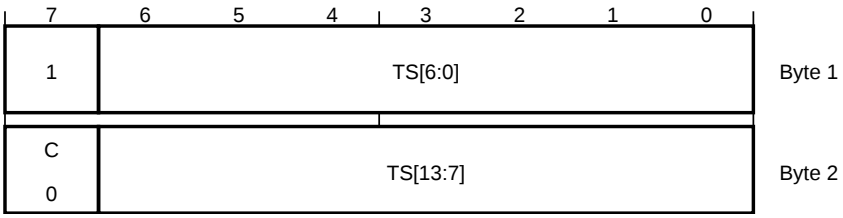
Other values encode different sizes of Hardware and Software source packets. This field reads as `0b00`.

### F1.2.11.2 Local Timestamp 1 packet payload

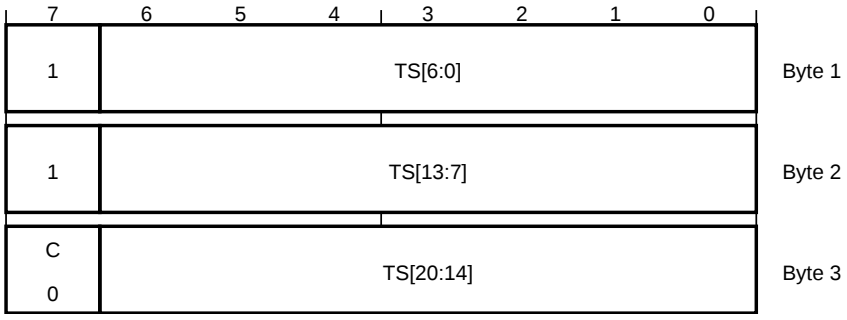
When 7-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



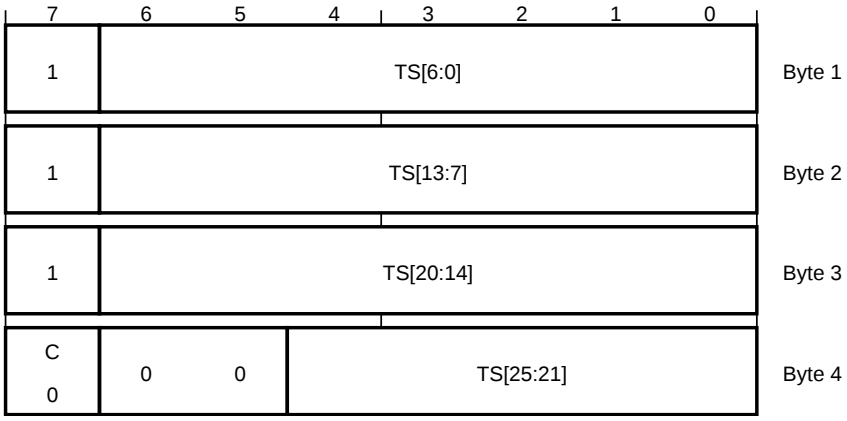

When 14-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:




When 21-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:




When 28-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:




**C, byte 4 bit [7], byte 3 bit [7], byte 2 bit [7], byte 1 bit [7]** Continuation bit. The defined values of this field are:

`0` Last byte of the packet.

`1` Another byte follows.

**Byte 4 bits [6:5], when 28-bit timestamp** This field reads-as-zero.

**TS, byte 4 bits [4:0], byte 3 bits [6:0], byte 2 bits [6:0], byte 1 bits [6:0]** Local Timestamp.

The timestamp is 28 bits. If the Local Timestamp 1 packet is shorter than 5 bytes, the most significant bits of the timestamp are zero.

### F1.2.12 Local Timestamp 2 packet

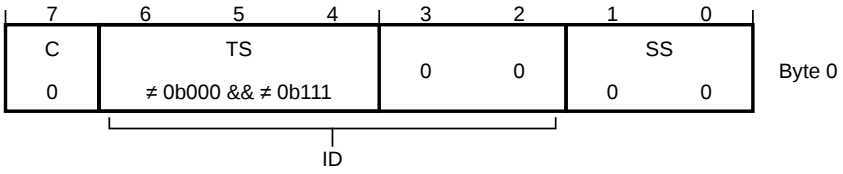
The Local Timestamp 2 packet characteristics are:

**Purpose** If the ITM outputs the Local Timestamp synchronously to the corresponding ITM or DWT data, and the required timestamp value is in the range 1-6, it uses the Local Timestamp 2 packet. For more information, see Local Timestamp 1 packet.

**Attributes** 8-bit Protocol packet.

#### Field descriptions

The Local Timestamp 2 packet bit assignments are:




**C, byte 0 bit [7]** Continuation bit. This bit reads as zero.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b00000** See Synchronization packet.

**0bxxx00** For all other values of 0bxxx. Local Timestamp 2 packet.

**0b11100** See Overflow packet.

This field reads as 0bxxx00.

**TS, byte 0 bits [6:4]** Local timestamp value, in the range 0b001 to 0b110.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.13 Overflow packet

The Overflow packet characteristics are:

**Purpose** The ITM outputs an Overflow packet if:

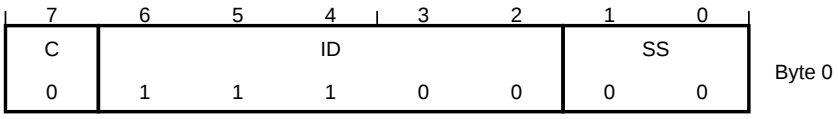
- Software writes to a Stimulus Port register when the stimulus port output buffer is full.
- The DWT attempts to generate a Hardware source packet when the DWT output buffer is full.
- The Local timestamp counter overflows.

The Overflow packet comprises a header with no payload.

**Attributes** 8-bit Protocol packet.

#### Field descriptions

The Overflow packet bit assignments are:




**C, byte 0 bit [7]** Continuation bit. This bit reads as zero.

**ID, byte 0 bits [6:2]** Protocol packet type. Bits [6:2] discriminate between Protocol packet types. The defined values of this field are:

**0b11100** Overflow packet.

This field reads as 0b11100.

**SS, byte 0 bits [1:0]** Packet type. The defined values of this field are:

**0b00** Protocol packet.

Other values encode different sizes of Hardware and Software source packets. This field reads as 0b00.

### F1.2.14 Periodic PC Sample packet

The Periodic PC Sample packet characteristics are:

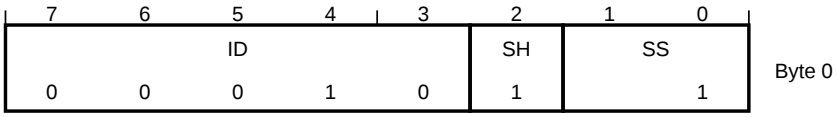
**Purpose** The DWT unit generates PC samples at fixed time intervals, with an accuracy of one clock cycle. The POSTCNT counter period determines the PC sampling interval. Software configures the DWT_CTRL.CYCTAP and DWT_CTRL.POSTINIT fields to determine how POSTCNT relates to DWT_CYCCNT. The DWT_CTRL.PCSAMPLENA bit enables PC sampling.

**Attributes** Multi-part Hardware source packet comprising:

- 8-bit header.
- 8 or 32-bit payload.

#### F1.2.14.1 Periodic PC Sample packet header

The Periodic PC Sample packet header bit assignments are:




**ID, byte 0 bits [7:3]** Discriminator ID. The defined values of this field are:

**0b00010** Periodic PC Sample packet.

This field reads as 0b00010.

**SH, byte 0 bit [2]** Source. The defined values of this bit are:

**1** Hardware source packet.

This bit reads as one.

**SS, byte 0 bits [1:0]** Size. The defined values of this field are:

**0b01** Source packet, 1-byte payload, 2-byte packet.

**0b11** Source packet, 4-byte payload, 5-byte packet.

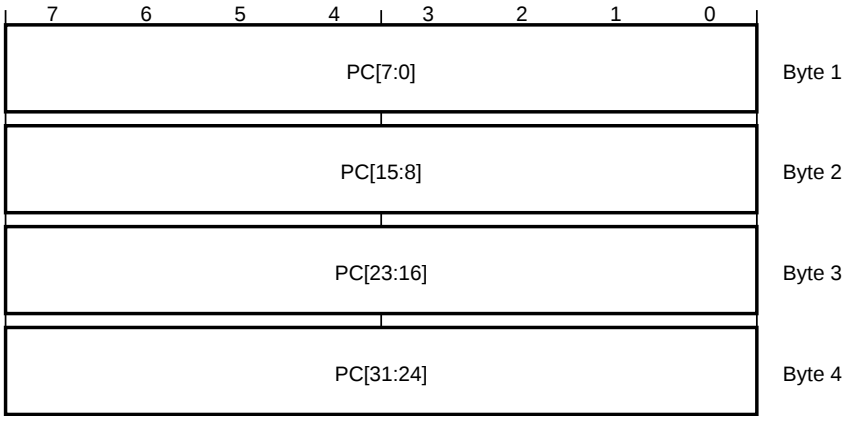
SS == 0b10 is invalid for a Periodic PC Sample packet.

The value 0b00 encodes a Protocol packet.

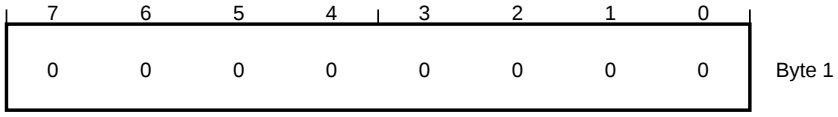
This field reads as 0bx1.

#### F1.2.14.2 Periodic PC Sample packet payload

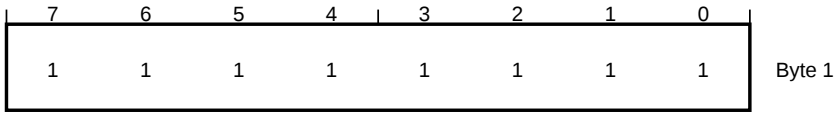
When Allowed and not sleeping, SS == 0b11, the Periodic PC Sample packet payload bit assignments are:




When Allowed and sleeping, SS == `0b01`, the Periodic PC Sample packet payload bit assignments are:




When Prohibited, SS == `0b01`, the Periodic PC Sample packet payload bit assignments are:




**PC, bytes <4:1>, when Allowed and not sleeping, SS == `0b11`** Periodic PC sample value.

**Byte <1>, when Allowed and sleeping, SS == `0b01`** This field reads as `0b00000000`.

**Byte <1>, when Prohibited, SS == `0b01`** This field reads as `0b11111111`.

### F1.2.15 PMU overflow packet

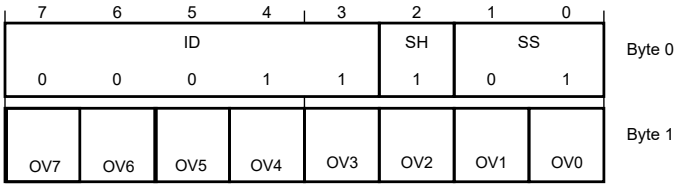
The PMU overflow packet characteristics are:

**Purpose** For each counter *n*, if the lower eight bits of that counter overflow, the associated OV*n* of the PMU overflow packet is set. If multiple counters overflow in the same period, multiple bits might be set. If there are fewer than 8 general-purpose counters, the associated PMU overflow packet bit is always zero.

**Attributes** 8 bit protocol packet.

#### Field descriptions

The PMU overflow packet bit assignments are:




**Byte 1 bits [7:0]** Packet Header

**Byte 2 bits [7:0]** OV*n*

## F1.2.16 Synchronization packet

The Synchronization packet characteristics are:

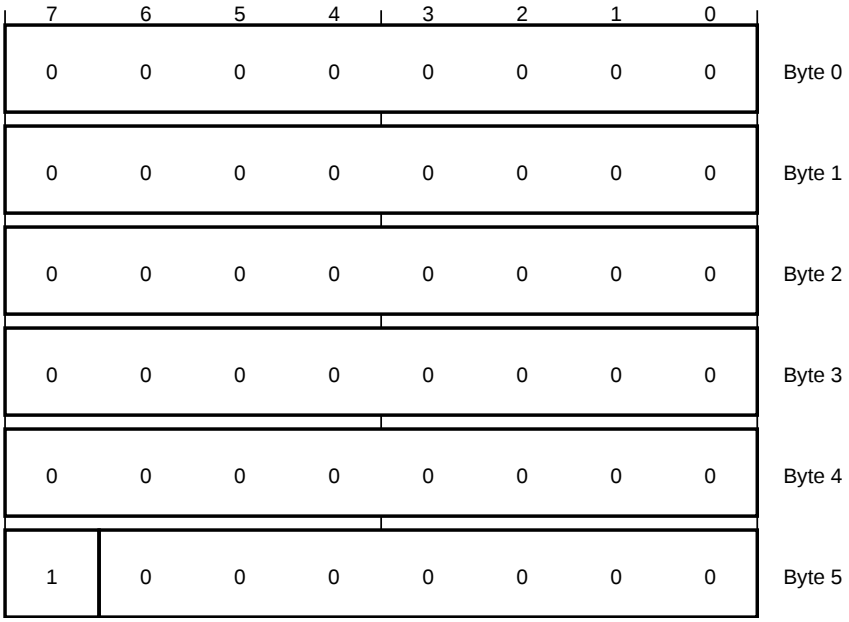
**Purpose** A Synchronization packet provides a unique pattern in the bit stream. Trace capture hardware can identify this pattern and use it to identify the alignment of packet bytes in the bitstream.

**Attributes** 48-bit Protocol packet.

A Synchronization packet is at least forty-seven 0 bits followed by single 1 bit. This section describes the smallest possible Synchronization packet.

### Field descriptions

The Synchronization packet bit assignments are:




**Byte 5 bit [7]** Indicates the end of the Synchronization packet. This bit reads as one.

**Byte 5 bits [6:0], bytes <4:1>** This field reads-as-zero.

**Byte <0>** This field reads as `0b00000000`.

# Glossary

### AAPCS

Procedure Call Standard for the Arm Architecture.

### Address dependency

An address dependency exists when the value that is returned by a read computes the address of a subsequent access. An address dependency exists even if the value that is returned by the first read does not change the address of the second read or write.

### Addressing mode

Means a method for generating the memory address that is used by a load/store instruction.

### Aligned

A data item that is stored at an address that is exactly divisible by the highest power of 2 that divides exactly into its size in bytes. Aligned halfwords, words, and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of each element of the access.

### Application Program Status Register (APSR)

The register containing those bits that deliver status information about the results of instructions, the N, Z, C, and V bits of the XPSR. In an implementation that includes the DSP extension, the APSR includes the GE bits that provide status information from DSP operations.

*See also* B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

### APSR

*See* Application Program Status Register.

### Architecturally executed

An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

In a PE that performs Speculative execution, an instruction is not architecturally executed if the PE discards the results of a Speculative execution.

*See also* Condition code check, Simple sequential execution.

### Architecturally Unknown

An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

*See also* Implementation Defined.

### Architecture tick

An atomic unit of execution. In the Armv8.0-M architecture, most instructions are considered atomic units for execution (they are either performed or not performed). The most notable exceptions are instructions that support ICI behavior.

### Associativity

*See* Cache associativity

**Atomicity**

Describes either single-copy atomicity or multi-copy atomicity. B6.5 *Atomicity* on page 191 defines these forms of atomicity for the Arm architecture.

*See also* Multi-copy atomicity, Single-copy atomicity.

**Attributability**

A PMU event that is caused by the PE counting the PMU event is Attributable. If an agent other than the PE that is counting the PMU events causes a PMU event then that PMU event is Unattributable.

A PMU event is either Attributable or Unattributable. If the PMU event is Attributable, it is further defined whether the PMU event is Attributable to:

- The current Security state of the PE.
- The privilege level.
- When the PE is in Debug state, operations issued to the PE by the Debugger through the external debug interface.

**Attribution Unit (AU)**

The combination of the Secure Attribution Unit (SAU) and the Implementation Defined Attribution Unit (IDAU).

*See also* Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

**AU**

*See* Attribution unit.

**Availability**

Readiness for correct service.

**Background state**

The state of the PE before the last (previous) preemption occurred.

**Banked register**

A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

**Base register**

A register that is specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the address that is sent to memory.

**Base register Write-Back**

Describes writing back a modified value to the base register used in an address calculation.

**Baseboard Management Controller**

A PE dedicated to system control and monitoring.

**Beat**

The execution of a 1/4 of an MVE vector operation. Because the vector length is 128 bits, one beat of a vector add instruction equates to computing 32 bits of result data. This is independent of lane width. For example, if a lane width is 8 bits, then a single beat of a vector add instruction would perform four 8-bit additions.

*See also* B5.4 *Beats* on page 170.

**Behaves as if**

Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

**BF branch point**

The interstice between two instructions, that is the gap between the instruction that immediately precedes the instruction at the branch target of a BF instruction and the instruction that is identified by the branch target of a BF instruction.

*See also* B3.29 *Branch future* on page 132.

**Big-endian memory**

Means that, for example:

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

*See also* B6.3 *Endianness* on page 188, Little-endian memory.

**Blocking**

Describes an operation that does not permit following instructions to be executed before the operation completes.

A non-blocking operation can permit following instructions to be executed before the operation completes, and in the event of encountering an exception does not signal an exception to the PE. This enables implementations to retire following instructions while the non-blocking operation is executing, without the need to retain precise PE state.

**Branch prediction**

Is where a PE selects a future execution path to fetch along. For example, after a branch instruction, the PE can choose to speculatively fetch either the instruction following the branch or the instruction at the branch target.

*See also* Prefetching.

**Breakpoint**

A debug event that is triggered by the execution of a particular instruction, which is specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

**Byte**

An 8-bit data item.

**Cache associativity**

The number of locations in a cache set to which an address can be assigned. Each location is identified by its *way* value.

**Cache level**

The position of a cache in the cache hierarchy. In the Arm architecture, the lower numbered levels are those closest to the PE. For more information, see B6.24 *Caches* on page 225.

**Cache line**

The basic unit of storage in a cache. Its size in words is always a power of two, usually four or eight words. A cache line must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely called cache lines.

**Cache sets**

Areas of a cache, which is divided up to simplify and speed up the process of determining whether a cache hit occurs. The number of cache sets is always a power of two. The term cache sets is a common convention for describing cache memories, and this description must not be treated as defining a property of the cache.

**Cache way**

A cache way consists of one cache line from each cache set. The cache ways are indexed from 0 to (Associativity-1). Each cache line in a cache way is chosen to have the same index as the cache way. For example, cache way *n* consists of the cache line with index *n* from each cache set. The term cache way is a common convention for describing cache memories, and this description must not be treated as defining a property of the cache.

**Cache write-back granule**

The maximum size of the memory that can be overwritten. In some implementations, the CTR identifies the Cache Write-Back Granule.

**Callee-saved registers**

Are registers that a called procedure must preserve. To preserve a callee-saved register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and reload it from the stack during procedure exit.

**Caller-saved registers**

Are registers that a called procedure is not required to preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

**Catastrophic failure**

A failure with harmful consequences that are orders of magnitude, or even incommensurably, higher than the benefit provided by correct service delivery.

**Chained vector instruction**

An instruction that is subject to beat-wise execution.

**Coherence order**

*See* Coherent

**Coherent**

Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

**Condition code check**

The process of determining whether a conditional instruction executes normally or is treated as a NOP. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of EPSR.IT determines whether the instruction is executed normally.

*See also* Condition code field, Condition flags, Conditional execution.

**Condition code field**

A 4-bit field in an instruction that specifies the condition under which the instruction executes.

*See also* Condition code check.

**Condition flags**

The N, Z, C, and V bits of APSR, or XPSR. See B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73 for more information.

*See also* Condition code check.

### Conditional execution

When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`. See C1.3 *Conditional execution* on page 429.

*See also* Condition code check.

### Configuration

Settings that are made on reset, or immediately after reset, and normally expected to remain static throughout program execution.

### CONSTRAINED UNPREDICTABLE

Where an instruction can result in UNPREDICTABLE behavior, the Armv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior within the limits defined for each particular case, and this behavior might vary.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALLCAPS.

*See also* Unpredictable.

### Containable

An error that is not uncontained. A Containable error is also referred to as a Contained error.

### Context switch

The saving and restoring of computational state when switching between different threads or processes. In this manual, the term context switch describes any situation where the context is switched by an operating system and might or might not include changes to the address space.

### Context synchronization event

A context synchronization event is one of the following:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Taking an exception.
- Returning from an exception.
- Exit from Debug state.

For more information, see B3.35 *Context Synchronization Event* on page 148.

**Note**

Security state transitions are not Context synchronization events.

### Control dependency

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

### Corrected

An error that is detected by hardware and that hardware can correct. This is also referred to as a Correctable error.

### Cross beat

An operation that requires operands from different beats to produce the output for a single beat. For example, a vector widening operation would be cross beat, whereas a vector addition would not be.

### Cross Trigger Interface

A debug component that is not part of the Armv8-M architecture.

**CTI**

See Cross Trigger Interface.

**DAP**

Debug Access Port.

**Data independent timing**

The time that it takes to execute a piece of code where the time is not a function of the data being operated on.

*See also* B3.34 *Data independent timing* on page 145.

**Data Watchpoint and Trace (DWT)**

The Data Watchpoint and Trace unit is a component of Armv8-M debug that optionally provides a number of trace, sampling, and profiling functions.

*See also* B13.2 *Data Watchpoint and Trace unit* on page 334.

**DCB**

*See* Debug Control Block.

**Debug Control Block (DCB)**

A region in the System Control Space that is assigned to registers that support debug features.

*See also* System Control Space.

**Debugger**

In most of this manual, *debugger* refers to any agent that is performing debug. However, some parts of the manual require a more rigorous definition, and define debugger locally. See Chapter B12 *Debug* on page 273.

**Deferred**

An error that has not been silently propagated but does not require immediate action at the producer. The error might have passed from the producer to the consumer.

**Deprecated**

Something that is present in the Arm architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

*See also* OPTIONAL.

**Detected**

An error that has been detected and signaled to a consumer.

**Detected Uncorrectable**

A detected error that cannot be corrected and causes failure.

**Digital signal processing (DSP)**

Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

**Direct access**

A read or write of a register.

**DIT**

*See* Data independent timing.

### Domain

In the Arm architecture, *domain* is used in the following contexts.

**Shareability domain** Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

**Power domain** Defines a block of logic with a single, common, power supply.

### Double-precision value

Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

### Doubleword

A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

### Doubleword-aligned

Means that the address is divisible by 8.

### DSP

*See* Digital signal processing.

### DWT

*See* Data Watchpoint and Trace.

### ECC

Error Correction Code

### EDC

Error Detection Code

### Effective value

A register control field, meaning a field in a register that controls some aspect of the behavior, can be described as having an *Effective value*:

```
* In some cases, the description of a particular control *a* specifies that when
control *a* is active it causes a register control field *b* to be treated
as having a fixed value for all purposes other than direct reads, or
direct reads and direct writes, of the register containing control field *b*.
When control *a* is active that fixed value is described as the *Effective
value* of register control field *b*.

In other cases, a register control field *b* is not
implemented or is not accessible, but behavior of the PE is as if control
field *b* was implemented and accessible, and had a particular value.
In this case, that value is the *Effective value* of register control field *b*.

Where a register control field is introduced in a particular version of
the architecture, and is not implemented in an earlier version of the
architecture, typically it will have an *Effective value* in that earlier
version of the architecture.

* Otherwise, the *Effective value* of a register control field is the value
of that field.
```

### Element

The data that is put into a lane.

*See also* Lane.

**Embedded Trace Macrocell (ETM)**

A component of the Arm CoreSight debug and trace solution. An ETM provides non-invasive trace of PE operation.

**Endianness**

An aspect of the system memory mapping. For more information, see B6.3 *Endianness* on page 188.

*See also* Big-endian memory and Little-endian memory.

**EPSR**

See Execution Program Status Register.

**Error**

Deviation from correct service or a correct value.

**Error propagation**

Passing an error from a producer to a consumer.

**Error record**

Data recorded about an error, usually by hardware.

**ETM**

*See* Embedded Trace Macrocell

**Exception**

Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

**Exception vector**

A fixed address that contains the address of the first instruction of the corresponding exception handler.

**Execution Program Status Register (EPSR)**

A register that contains the Execution state bits and is part of the XPSR.

*See also* B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

**Execution stream**

The stream of instructions that would have been executed by sequential execution of the program.

**Explicit access**

A read from memory, or a write to memory, generated by a load or store instruction that is executed by the PE.

**Failure**

The event of deviation from correct service.

**Fault**

An exception that is generated because of some form of system error. A BusFault might be a RAS fault.

**Fault injection**

The deliberate injection of faults into a system for testing.

**Fault prevention**

Designing a system to avoid faults.

**Fault removal**

Logic or other mechanisms for detecting faults and correcting or bypassing their effect.

**Field Replaceable Unit**

The smallest unit that can be replaced without return to base.

**Flash Patch and Breakpoint Unit**

The Flash Patch and Breakpoint unit supports setting breakpoints on instruction fetches.

*See also* B13.5 *Flash Patch and Breakpoint unit* on page 359.

**Flush-to-zero mode**

A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

**FPB**

*See* Flash Patch and Breakpoint Unit.

**FRU**

*See* Field Replaceable Unit.

**General-purpose registers**

The registers that the base instructions use for processing:

- The general-purpose registers are R0-R12. R13-R14 are the SP and LR, respectively. For more information, see B3.3 *Registers* on page 70.

*See also* High registers, Low registers.

**Halfword**

A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

**Halfword-aligned**

Means that the address is divisible by 2.

**Hardware fault**

A fault that originates in or affects hardware.

**High registers**

The general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

**Note**

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

*See also* General-purpose registers, Low registers.

**ICI**

*See* Interrupt continuable instruction.

**If-Then block (IT block)**

An IT block is a block of up to four instructions following an *If-Then* (IT) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some are the inverse of others.

**Immediate and offset fields**

Are unsigned unless otherwise stated.

**Immediate value**

A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many T32 instructions can be used with an immediate argument.

**IMP DEF**

An abbreviation that is used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

**IMPLEMENTATION DEFINED**

Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALLCAPS.

**Implicit access**

An access that is not explicit.

*See also* Explicit access.

**Imprecise exception**

An exception that is generated as the result of a system error. An imprecise exception is reported at the time that is asynchronous to the instruction that caused it.

**Index register**

A register that is specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

**Indirect access**

A read or write of a register that is not a direct access.

For example, an indirect write to a register might occur as the side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation that is performed by an external agent.

*See also* Direct access

**Infected**

Being in error.

**Inline literals**

These are constant addresses and other data items that are held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

**Instrumentation Trace Macrocell (ITM)**

A component of the Arm CoreSight debug and trace solution. An ITM provides a memory-mapped register interface that applications can use to write logging or event words to a trace sink.

**Interrupt continuable instruction**

Instructions that can be interrupted part way through their execution. After the interrupt service routine has completed, execution of the partly executed instruction can be resumed and the instruction is not required to be restarted from the beginning.

**Interrupt Program Status Register (IPSR)**

The register that provides status information on whether an application thread or exception handler is executing on the processor. If an exception handler is executing, the register provides information on the exception type. The register is part of the XPSR.

*See also* B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

**Interrupt Service Routine**

The procedure that handles an interrupt.

**Interworking**

A method of working that permits branches between software using the A32 and T32 instruction sets in the Armv8-A architecture. For Armv8-M, interworking is described in C1.4.7 *Instruction set, interworking and interstating support* on page 439.

**IPSR**

*See* Interrupt Program Status Register.

**Isolation**

Limiting the impact of an error only to components that actually try to use corrupted data.

**ISR**

*See* Interrupt Service Routine.

**ITM**

*See* Instrumentation Trace Macrocell.

**Lane**

A section of a vector register or operation.

**Latent fault**

An error that is present in a system but not yet detected.

**Level**

*See* Cache level.

**Level of Coherence (LoC)**

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of coherency.

*See also* Cache level, Point of Coherency.

**Level of Unification, Inner Shareable (LoUIS)**

The last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for the Inner Shareable Shareability domain.

*See also* Cache level, Point of Unification.

**Level of Unification, uniprocessor (LoUU)**

For a PE, the last level of cache that must be cleaned or invalidated when cleaning or invalidating to the point of unification for that PE.

*See also* Cache level, Point of Unification.

**Line**

*See* Cache line.

**Little-endian memory**

Means that, for example:

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

*See also* Big-endian memory, B6.3 *Endianness* on page 188.

**Load/store architecture**

An architecture where data-processing operations only operate on register contents, not directly on memory contents.

**LoC**

*See* Level of Coherence.

**Lockup**

A PE state where the PE stops executing instructions in response to an error for which escalation to an appropriate HardFault handler is not possible because of the current execution priority. For more information, see B3.33 *Lockup* on page 139.

**LoUIS**

See Level of Unification, Inner Shareable.

**LoUU**

*See* Level of Unification, uniprocessor.

**Low registers**

General-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

**LR hazard**

An LR hazard occurs when LR is written while another instruction, other than `LE` or `LETP`, is accessing LR.

**M-profile Vector Extension**

An optional part of the Armv8.1-M architecture that supports both integer and floating-point data types.

**Memory barriers**

The term memory barrier is the general term that is applied to an instruction, or sequence of instructions, that forces synchronization events by a PE regarding retiring Load/Store instructions. For more information, see B6.13 *Memory barriers* on page 203.

**Memory coherency**

The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value that is actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

**Memory hint**

A memory hint instruction provides advance information to memory systems about future memory accesses, without actually loading or storing any data to or from the register file. PLD and PLI are the only memory hint instructions that are defined in Armv8-M.

**Memory Protection Unit (MPU)**

A hardware unit whose registers provide simple control of a limited number of protection regions in memory, for more information, see Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

**Minor failure**

A failure with harmful consequences that are of a similar cost to the benefits that are provided by correct service delivery.

**MPU**

*See* Chapter B9 *The Armv8-M Protected Memory System Architecture* on page 257.

**Multi-copy atomicity**

The form of atomicity that is described in B6.5.2 *Multi-copy atomicity* on page 191.

*See also* Atomicity, Single-copy atomicity.

**MVE**

*See* M-profile Vector Extension.

**NaN**

Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the IEEE Standard for Floating-point Arithmetic.

**Node**

A component that detects an error is called a node.

**Non-Return-to-Zero (NRZ)**

A physical layer signaling scheme that is used on asynchronous communication ports

**NRZ**

*See* Non-Return-to-Zero.

**Observer**

A master in the system that is capable of observing memory accesses. For more information, see B6.8 *Observability of memory accesses* on page 197.

**Obsolete**

Obsolete indicates something that is no longer supported by Arm. When an architectural feature is described as obsolete, this indicates that the architecture has no support for that feature, although an earlier version of the architecture did support it.

**Offset addressing**

Means that the memory address is formed by adding or subtracting an offset to or from the base register value.

**OPTIONAL**

When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a feature to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALLCAPS.

Note: Do not confuse these Arm-specific uses of OPTIONAL with other uses of OPTIONAL, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior that is determined by an implementation choice.

*See* also Deprecated.

**PE**

*See* Processing element.

**Performance Monitoring Unit**

An optional non-invasive debug component that allows events to be identified and counted.

**Persistent fault**

A fault that is not transient.

**Physical address (PA)**

An address that identifies a location in the physical memory map.

**PMU**

*See* Performing Monitoring Unit.

**PoC**

*See* Point of Coherency.

**Point of coherency (PoC)**

For a particular MVA, the point at which all agents that can access memory are guaranteed to see the same copy of a memory location.

**Point of unification (PoU)**

For a particular PE, the point by which the instruction and data caches of that PE are guaranteed to see the same copy of a memory location.

**Poisoned**

State that has been marked as being in error so that subsequent consumption of the state signals a detected error to a consumer.

**Post-indexed addressing**

Means that the memory address is the base register value, but an offset is added to or subtracted from the base register value and the result is written back to the base register.

**PoU**

*See* Point of Unification.

**PPB**

Private Peripheral Bus

**Pre-indexed addressing**

Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

**Prefetching**

Prefetching refers to speculatively fetching instructions or data from the memory system. In particular, instruction prefetching is the process of fetching instructions from memory before the instructions that precede them, in simple sequential execution of the program, have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

In this manual, references to instruction or data fetching apply also to prefetching, unless the context explicitly indicates otherwise.

*See also* Simple sequential execution.

**Privileged access**

Memory systems typically differentiate between privileged and unprivileged accesses, and support more restrictive permissions for unprivileged accesses. Some instructions can be used only by privileged software.

**Processing element (PE)**

The abstract machine that is defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

**Program Status Registers (XPSR)**

XPSR is the term that is used to describe the combination of the APSR, EPSR, and IPSR into a single 32-bit Program Status Register.

*See also* B3.5 *XPSR, APSR, IPSR, and EPSR* on page 73.

**Protection granule**

A quantum of memory for which an EDC or ECC provides detection or correction.

**Protection region**

A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

**Protection Unit**

*See* Memory Protection Unit

**Pseudo-instruction**

UAL assembler syntax that assembles to an instruction encoding that is expected to disassemble to a different assembler syntax, and is described in this manual under that other syntax. For example, `MOV<Rd>, <Rm>, LSL #<n>` is a pseudo-instruction that is expected to disassemble as `LSL<Rd>, <Rm>, #<n>`.

*See also* Chapter C1 *Instruction Set Overview* on page 420.

**Quadword**

A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

**Quadword-aligned**

Means that the address is divisible by 16.

**Quiet NaN**

A NaN that propagates unchanged through most floating-point operations.

**RAO**

*See* Read-As-One.

**RAO/SBOP**

In versions of the Arm architecture before Armv8, Read-As-One, Should-Be-One-or-Preserved on writes.

In Armv8, RES1 replaces this description.

*See also* UNK/SBOP, Read-As-One, RES1, Should-Be-One-or-Preserved (SBOP).

**RAO/WI**

Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software can rely on the field reading as all 1s, and on writes being ignored.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

*See also* Read-As-One.

**RAS**

Reliability, Availability, and Serviceability.

**RAZ**

*See* Read-As-Zero.

**RAZ/SBZP**

In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In Armv8, RES0 replaces this description.

*See also* UNK/SBZP, Read-As-Zero, RES0, Should-Be-Zero-or-Preserved (SBOP).

**RAZ/WI**

Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-Zero, and must ignore writes to the field.

Software can rely on the field reading as all 0s, and on writes being ignored.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

*See also* Read-As-Zero.

**Read, modify, write**

In a read, modify, write instruction sequence, a value is read to a general-purpose register, the relevant fields that are updated in that register, and the new value that is written back.

**Read-allocate cache**

A cache in which a cache miss on reading data causes a cache line to be allocated into the cache.

**Read-As-One (RAO)**

Hardware must implement the field as reading as all 1s.

Software:

- Can rely on the field reading as all 1s.
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 1, or to a field that reads as all 1s. It applies only to a bit or field that is read-only.

*See also* RAO/SBOP, RAO/WI, RES1.

**Read-As-Zero (RAZ)**

Hardware must implement the field as reading as all 0s.

Software:

- Can rely on the field reading as all 0s
- Must use a SBOP policy to write to the field.

This description can apply to a single bit that reads as 0, or to a field that reads as all 0s. It applies only to a bit or field that is read-only.

*See also* RAZ/SBZP, RAZ/WI, RES0.

**Recoverable**

A contained error that must be corrected to allow the correct operation of the system or smaller parts of the system to continue.

*See also* B15.3 *Generating error exceptions* on page 405.

### Register data dependency

A register data dependency exists between a first data value and a second data value when either:

- The register that holds the first data value is used in the calculation of the second data value, and the calculation between the first data value and the second data value does not consist of either:
  - A conditional branch whose condition is determined by the first data value.
  - A conditional selection, move, or computation whose condition is determined by the first data value, where the input data values for the selection, move, or computation do not have a data dependency on the first data value.
- There is a register data dependency between the first data value and a third data value, and between the third data value and the second data value.

### Reliability

Continuity of correct service.

### RES0

A reserved bit or field with Should-Be-Zero-or-Preserved behavior, or equivalent read-only or write-only behavior. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

**Note**

RES0 is not used in descriptions of instruction encodings.

This means the definition of RES0 for fields in read/write registers is:

**If a bit is RES0 in all contexts**

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value that is successfully written, by either a direct or an indirect write, to the bit.

   If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location that is associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

**If a bit is RES0 only in some contexts**

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.

- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.
  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that is written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALLCAPS.

*See also* Read-As-Zero, RES1, Should-Be-Zero-or-Preserved, UNKNOWN.

### RES0H

A reserved bit or field with Should-Be-Zero-or-Preserved (SBZP). This behavior uses the *Hardwired to 0* subset of the RES0 definition.

### RES1

A reserved bit or field with Should-Be-One-or-Preserved behavior, or equivalent read-only or write-only behavior. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

**Note**

RES1 is not used in descriptions of instruction encodings.

This means the definition of RES1 for fields in read/write registers is:

**If a bit is RES1 in all contexts**

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.

- A read of the bit returns the last value that is successfully written, by either a direct or an indirect write, to the bit.
  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

**If a bit is RES1 only in some contexts**

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  **Note**

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that is written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALLCAPS.

*See also* Read-As-One, RES0, Should-Be-One-or-Preserved, UNKNOWN.

### RES1H

A reserved bit or field with Should-Be-One-or-Preserved (SBOP) behavior. This behavior uses the *Hardwired to 1* subset of the RES1 definition.

### Reserved

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions that are described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

*See also* CONSTRAINED UNPREDICTABLE, RES0, RES1, UNDEFINED, UNK, UNPREDICTABLE.

**Restartable**

A contained error that does not immediately impact correct operation. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part.

*See also* B15.3 *Generating error exceptions* on page 405.

**Return Link**

A value relating to the return address.

**RISC**

Reduced Instruction Set Computer.

**Rounding error**

The value of the rounded result of an arithmetic operation minus the exact result of the operation.

**Rounding mode**

Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic*.

**Saturated arithmetic**

Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31} - 1$ to $-2^{31}$ or the opposite way.

**SBO**

*See* Should-Be-One.

**SBOP**

*See* Should-Be-One-or-Preserved.

**SBZ**

*See* Should-Be-Zero.

**SBZP**

*See* Should-Be-Zero-or-Preserved

**Security hole**

A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTALBE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Self-modifying code**

Code that writes one or more instructions to memory and then executes them. When using self-modifying code, cache maintenance and barrier instructions must be used to ensure synchronization.

**Serial Wire Output (SWO)**

An asynchronous TPIU port supporting one or both of the NRZ and Manchester encodings.

**Serial Wire Viewer (SWV)**

The combination of an SWO and at least one of a DWT unit or an ITM, providing data tracing capability.

**Service failure mode**

A mode entered to reduce the severity of an error.

**Serviceability**

The ability to undergo modification and repairs.

**Set**

*See* Cache sets.

**Should-Be-One (SBO)**

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-One-or-Preserved (SBOP)**

From the introduction of the Armv8 architecture, the description *Should-Be-One-0r -Preserved* is superseded by RES1.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 1s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-Zero (SBZ)**

Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Should-Be-Zero-or-Preserved (SBZP)**

From the introduction of the Armv8 architecture, the description *Should-Be-Zero -or-Preserved* is superseded by RES0.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

**Signaling NaNs**

Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

**Signed data types**

Represent an integer in the range $-2^{N-1}$ to $+2^{N-1} - 1$, using two's complement format.

**Signed immediate and offset fields**

Are encoded in two's complement notation unless otherwise stated.

**Silent data corruption**

An error that is not detected by hardware or software.

**Silently propagated**

An error that is passed from place to place without being signaled as a detected error.

**SIMD**

Single-Instruction, Multiple-Data.

**Simple sequential execution**

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no Speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

**Single peripheral**

A single peripheral is a region of memory of an IMPLEMENTATION DEFINED size that is defined by the peripheral.

**Single-copy atomicity**

The form of atomicity that is described in B6.5.1 *Single-copy atomicity* on page 191.

See also Atomicity, Multi-copy atomicity.

**Single-precision value**

A 32-bit word that is interpreted as a basic single-precision floating-point number according to the IEEE Standard for Floating-point Arithmetic.

**Software fault**

A fault that originates in and affects software.

**Spatial locality**

The observed effect that after a program has accessed a memory location, it is likely to also access nearby memory locations in the near future. Caches with multi-word cache lines exploit this effect to improve performance.

**Special-purpose register**

One of a specified set of registers for which all direct and indirect reads and writes to the register appear to occur in program order relative to other instructions, without the need for any explicit synchronization. For more information, see B3.3 *Registers* on page 70.

**Speculative writes**

All of the following are Speculative writes:

- Writes generated by store instructions that appear in the Execution stream after a branch that is not architecturally resolved.
- Writes generated by store instructions that appear in the Execution stream after an instruction where a synchronous exception condition has not been architecturally resolved.
- Writes generated by conditional store instructions for which the conditions for the instruction have not been architecturally resolved.
- Writes generated by store instructions for which the data being written comes from a register that has not been architecturally committed.

**System Control Block (SCB)**

An address region in the System Control Space, which is used for key feature control and configuration that is associated with the exception model.

*See also* System Control Space.

**System Control Space (SCS)**

A region of the memory map that is reserved for system control and configuration registers.

*See also* Debug Control Block, B7.3 *The System Control Space (SCS)* on page 245.

**T32 instruction**

One or two halfwords that specify an operation to be performed by a PE. T32 instructions must be halfword-aligned. For more information, see Chapter C1 *Instruction Set Overview* on page 420.

T32 instructions were previously called Thumb instructions.

**Tail-chaining**

An optimization that removes unstacking and stacking operations. For more information, see B3.26 *Tail-chaining* on page 123.

**Temporal locality**

The observed effect that after a program has accesses a memory location, it is likely to access the same memory location again in the near future. Caches exploit this effect to improve performance.

**TPIU**

*See*Trace Port Interface Unit.

**Trace Port Interface Unit (TPIU)**

A component of the Arm CoreSight debug and trace solution. A TPIU provides an external interface for one or more trace sources in the processor implementation.

**Transient fault**

A fault that is not persistent.

**UAL**

*See* Unified Assembler Language.

**Unaligned**

An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

**Unaligned memory accesses**

Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

**Unallocated**

Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

*See also* CONSTRAINED UNPREDICTABLE, UNPREDICTABLE, UNDEFINED.

**Uncontainable**

An error that has been, or might have been, silently propagated. This is also referred to as an Uncontained error.

*See also* B15.3 *Generating error exceptions* on page 405.

**UNDEFINED**

Indicates an instruction that generates an Undefined Instruction exception.

In body text, the term UNDEFINED is shown in SMALLCAPS.

*See also* Chapter C1 *Instruction Set Overview* on page 420.

**Undetected fault**

*See* Latent fault.

**Unified Assembler Language**

The assembler language that is introduced with Thumb-2 technology that is used in this manual. See Chapter C1 *Instruction Set Overview* on page 420 for details.

**Unified cache**

Is a cache that is used for both processing instruction fetches and processing data loads and stores.

**Unindexed addressing**

Means addressing in which the base register value is used directly as the address to send to memory, without adding or subtracting an offset. In most types of load/store instruction, unindexed addressing is performed by using offset addressing with an immediate offset of 0.

In the M-profile, the LDC, LDC2, STC, and STC2 instructions have an explicit unindexed addressing mode that permits the offset field in the instruction to specify additional coprocessor options.

**UNK**

An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

*See also* UNKNOWN.

**UNK/SBOP**

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

*See also* Read-as-One, Should-Be-One-or-Preserved, UNKNOWN.

**UNK/SBZP**

Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

*See also* Read-as-Zero, Should-Be-Zero-or-Preserved, UNKNOWN.

**UNKNOWN**

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using instructions that are not UNKNOWN, are not CONSTRAINED UNPREDICTABLE, and do not return UNKNOWN values.

An Unknown value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALLCAPS.

*See also* CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

**UNPREDICTABLE**

Means the behavior cannot be relied on. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege or security using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In body text, the term UNPREDICTABLE is shown in SMALLCAPS.

*See also* CONSTRAINED UNPREDICTABLE, UNDEFINED.

**Unrecoverable**

A contained error that is not recoverable. Continued correct operation is generally not possible. Usually this means correct operation of the system, but it can also be used in other contexts to describe correct operation of a smaller part. Systems might use high-level recovery techniques to work around an unrecoverable yet contained error in a component so that the system recovers from the error.

*See also* B15.3 *Generating error exceptions* on page 405.

**Unsigned data types**

Represent a non-negative integer in the range 0 to $+2^{N-1} - 1$, using normal binary format.

**Watchpoint**

A debug event that is triggered by an access to memory, which is specified in terms of the address of the location in memory being accessed.

**Way**

*See* Cache way.

**WI**

Writes Ignored. In a register that software can write to, a WI attribute that is applied to a bit or field indicates that the bit or field ignores the value that is written by software and retains the value it had before that write.

*See also* RAO/WI, RAZ/WI, RES0, RES1.

**Word**

A 32-bit data item. Words are normally word-aligned in Arm systems.

**Word-aligned**

Means that the address is divisible by 4.

**Write buffer**

A block of high-speed memory that optimizes stores to main memory.

**Write-Allocate cache**

A cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

**Write-back cache**

A cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or reallocated. Another common term for a write-back cache is a *copy-back cache*.

**Write-one-to-clear**

Writing 1 to the relevant bit clears it to 0. Writing 0 to the bit has no effect.

**Write-one-to-set**

Writing 1 to the relevant bit sets it to 0. Writing 0 to the bit has no effect.

**Write-Through cache**

A cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done using a write buffer, to avoid slowing down the PE.

**XPSR**

*See* Program Status Registers (XPSR)