Arm® v8-M Architecture Reference Manual

Arm DDI 0553A.g (ID121417)

Armv8-M Architecture Reference Manual

Release Information

The following releases of this document have been made.

Change History

Date	Issue	Confidentiality	Change
29 March 2016	A.a	Confidential - Beta	Beta release, limited circulation
28 July 2016	A.b	Non-confidential - Beta	Beta release
30 September 2016	A.c	Non-confidential - EAC	EAC release
30 November 2016	A.d	Non-confidential - EAC	Second EAC release
02 June 2017	A.e	Non-confidential - EAC	Third EAC release
29 September 2017	A.f	Non-confidential - EAC	Fourth EAC release
15 December 2017	A.g	Non-confidential - EAC	Fifth EAC release

The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

Proprietary Notice

Contents

Armv8-M Architecture Reference Manual

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 *About the Armv8 architecture, and architecture profiles* on page A1-24.

This manual has the following parts:

Part A	Provides an introduction to the Armv8-M architecture.
Part B	Describes the architectural rules.
Part C	Describes the T32 instruction set.
Part D	Describes the registers.
Part E	Describes the Armv8-M pseudocode.
Part F	Describes the packet protocols.

Using this book

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

Part A, Armv8-M Architecture Introduction and Overview

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

Chapter A1 ***Introduction***

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

Part B, Armv8-M Architecture Rules

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

Chapter B1 ***Resets***

Read this for a description of the reset rules.

Chapter B2 ***Power Management***

Read this for a description of the power management rules.

Chapter B3 ***Programmers' Model***

Read this for a description of the programmers model rules.

Chapter B4 ***Floating-point Support***

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

Chapter B5 ***Memory Model***

Read this for a description of the memory model rules.

Chapter B6 ***The System Address Map***

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

Chapter B7 ***Synchronization and Semaphores***

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

Chapter B8 ***The Armv8-M Protected Memory System Architecture***

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

Chapter B9 ***The System Timer, SysTick***

Read this for a description of the system timer rules.

Chapter B10 ***Nested Vectored Interrupt Controller***

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

Chapter B11 ***Debug***

Read this for a description of the debug rules.

Chapter B12 ***Debug and Trace Components***

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

Part C, Armv8-M Instructions

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

Chapter C1 ***Instruction Set Overview***

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

Chapter C2 ***Instruction Specification***

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

Part D, Armv8-M Registers

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

Chapter D1 ***Register Specification***

Read this for a description of the registers.

Part E, Armv8-M Pseudocode

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

Chapter E1 ***Arm Pseudocode Definition***

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

Chapter E2 ***Pseudocode Specification***

Read this for a description of the pseudocode.

Part F, Packet Protocols

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

Chapter F1 ***ITM and DWT Packet Protocol Specification***

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

Conventions

The following sections describe conventions that this book can use:

- *Typographic conventions*.
- *Signals*.
- *Numbers*.
- *Pseudocode descriptions* on page xvi.
- *Assembler syntax descriptions* on page xvi.

Typographic 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.

SMALL CAPITALS

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 `http://infocenter.arm.com`.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *A list of the assembler symbols for the instruction* on page C1-304.
- 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 ADC (immediate).

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 Part E *Armv8-M Pseudocode*.

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 *A list of the assembler symbols for the instruction* on page C1-304.

Additional reading

This section lists relevant publications from Arm and third parties.

See the Infocenter `http://infocenter.arm.com`, for access to Arm documentation.

Arm publications

- *ARM® Debug Interface v5 Architecture Specification* (Arm IHI 0031).
- *ARM® CoreSight™ Architecture Specification* (Arm IHI 0029).
- *ARM® Embedded Trace Macrocell Architecture Specification* ETMv4.0 to ETMv4.3 (Arm IHI 0064).
- *Embedded Trace Macrocell® ETMv1.0 to ETMv3.5 Architecture Specification* (Arm IHI 0014).
- *ARM®v6-M Architecture Reference Manual* (Arm DDI 0419).
- *ARM®v7-M Architecture Reference Manual* (Arm DDI 0403).
- *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* (Arm DDI 0487).

Other publications

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

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

  Note

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

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

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title.
- The number, Arm DDI 0553A.g.
- 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:

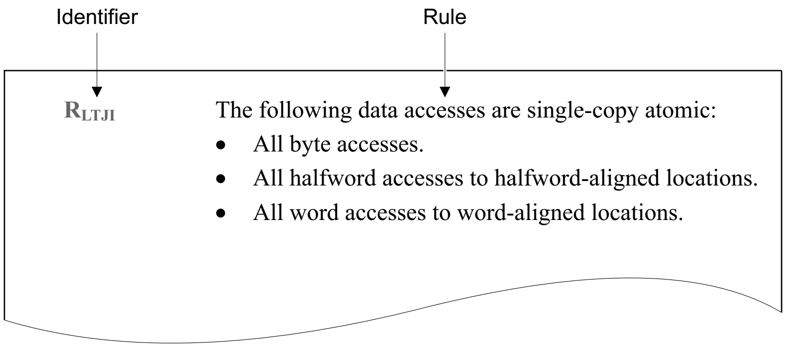
A1.1 Document layout and terminology

This section describes the structure and scope of, and the terminology that is used in, this manual. 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.


Identifier
Rule
R_{LTJI}
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.

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:

- Part C *Armv8-M Instruction Set*.
- Part D *Armv8-M Registers*.
- Part E *Armv8-M Pseudocode*.
- Part F *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 it 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 *Armv8-M variants* on page A1-27. 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 *Armv8-M variants* on page A1-27.

A1.3.2 MPU model

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

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 to 480 for external sources, and includes automatic vectoring and priority management, and automatic state preservation. See Chapter B10 *Nested Vectored Interrupt Controller*.

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 *Stack pointer* on page B3-51.

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*.

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

A1.3.6 Debug

The Armv8-M architecture introduces:

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

For more information about debug, see Chapter B11 *Debug* and Chapter B12 *Debug and Trace Components*.

A1.4 Armv8-M variants

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

DB The Debug Extension

Note

For details about the individual features that constitute the Debug Extension, see *Debug feature overview* on page B11-224.

DSP The Digital Signal Processing Extension.

A PE that implements the DSP Extension must implement the Main Extension.

FP The Floating-point Extension

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

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

M The Main Extension.

Note

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

MPU The Memory Protection Unit Extension

S The Security Extension

Note

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

ST The System Timer Extension

A table at the end of each section or subsection lists the extensions that an implementation must include in order for a particular rule to apply. Some extensions depend on the implementation of other extensions, for example FP.

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

R_{BDPL} There are two resets:

- Cold reset.
- Warm reset.

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

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

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

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

R_{WSZN} AIRCR.SYSRESETREQ is required to cause a Warm reset.

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

See also:

- Chapter B11 *Debug*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BDPL	From 8.0	None	-
CTPC	From 8.0	None	-
FNNX	From 8.0	None	-
GTXW	From 8.0	None	-
YMHN	From 8.0	None	-
WSZN	From 8.0	None	-
HFRS	From 8.0	None	-

Chapter B2
Power Management

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

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:

- Wait for Event (`WFE`).
- Wait For Interrupt (`WFI`).
- Sleep on exit (`SleepOnExit`).

B2.1.1 The Wait for Event (WFE) instruction

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

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

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

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

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

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

See also:

- *Priority model* on page B3-66.
- *WaitForEvent* on page E2-1352.
- *SendEvent* on page E2-1329.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DCMH	From 8.0	None	-
HDXV	From 8.0	None	-
KDND	From 8.0	None	-
YRDC	From 8.0	None	-

B2.1.2 The Event register

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

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

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

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

R_{CXMT} A reset clears the Event register.

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

See also:

- *SetEventRegister* on page E2-1330.
- *ClearEventRegister* on page E2-1221.
- *EventRegistered* on page E2-1240.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BPBR	From 8.0	None	-
CXMT	From 8.0	None	-

B2.1.3 The Wait for Interrupt (WFI) instruction

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

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

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

See also:

- *Priority model* on page B3-66.
- *WaitForInterrupt* on page E2-1352.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HRMJ	From 8.0	None	-

B2.1.4 Sleep on exit

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

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

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

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

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

See also:

- *Priority model* on page B3-66.
- *SleepOnExit* on page E2-1334.
- *Exception return* on page B3-87.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JXGW	From 8.0	None	-
CMVG	From 8.0	None	-
WWDW	From 8.0	None	-
LLQF	From 8.0	None	-

Chapter B3 Programmers' Model

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

- *Exceptions during exception return* on page B3-92.
- *Tail-chaining* on page B3-93.
- *Exceptions, instruction resume, or instruction restart* on page B3-95.
- *Vector tables* on page B3-98.
- *Hardware-controlled priority escalation to HardFault* on page B3-100.
- *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page B3-101.
- *Lockup* on page B3-103.
- *Context Synchronization Event* on page B3-110.
- *Coprocessor support* on page B3-111.

B3.1 PE modes, Thread mode and Handler mode

R_{CNMS} There are two PE modes:

- Thread mode.
- Handler mode.

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

Thread mode

Applications.

Handler mode

OS kernel and associated functions, that manage system resources.

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

R_{CMQP} Thread mode is selected on reset.

See also:

- *Privileged and unprivileged execution* on page B3-40.
- *Interrupt Program Status Register (IPSR)* on page B3-45.
- *Security states, Secure state, and Non-secure state* on page B3-49.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CNMS	From 8.0	None	-
RPKP	From 8.0	None	-
CMQP	From 8.0	None	-

B3.2 Privileged and unprivileged execution

R_{WVRK} **Thread mode**

Execution can be privileged or unprivileged.

Handler mode

Execution is always privileged.

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

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

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

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

See also:

- *PE modes, Thread mode and Handler mode* on page B3-39.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WVRK	From 8.0	None	-
SBQF	From 8.0	!M	-
JSSW	From 8.0	None	-

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.

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

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

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

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

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

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

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

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

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

R_{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 or return an UNKNOWN value for the source register that is specified as the PC.

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.

I_{QVWL} The choice between the behavior of the PC as a source or destination register might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

R_{LXPR} For instructions that specify two destination registers and if Rt, Rt2, RdLo, or RdHi 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.

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.

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.

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.

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.

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

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

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

See also:

- Chapter B6 *The System Address Map*.
- *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page B3-101.
- *Special-purpose CONTROL register* on page B3-44.
- *Stack limit checks* on page B3-84.
- *XPSR, APSR, IPSR, and EPSR* on page B3-45.
- *Resets, Cold reset, and Warm reset* on page B1-32.
- Chapter D1 *Register Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KGST	From 8.0	None	-
PLRT	From 8.0	M	-
QHMH	From 8.0	!M	-
PLNS	From 8.0	None	-
JPCB	From 8.0	None	-
XHHC	From 8.0	None	-
WMVJ	From 8.0	None	-
BGJG	From 8.0	None	-
LXPR	From 8.0	None	-
DRSS	From 8.0	None	-
XLVX	From 8.0	None	-
JFGT	From 8.0	None	-
XPBT	From 8.0	None	-
XPTQ	From 8.0	None	-
PDJC	From 8.0	None	-

B3.4 Special-purpose CONTROL register

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

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

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

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

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

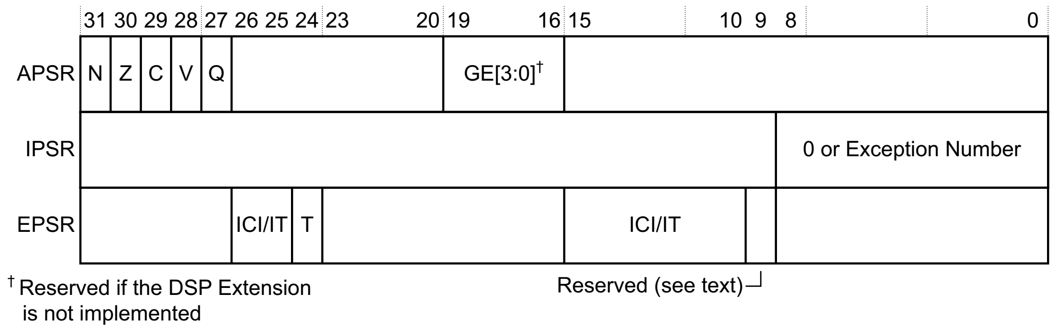
See also:
- *Context Synchronization Event* on page B3-110.
- *CONTROL, Control Register* on page D1-896.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CSPP	From 8.0	None	-
GKVQ	From 8.0	None	-
RJMP	From 8.0	None	-
HVGB	From 8.0	None	-

B3.5 XPSR, APSR, IPSR, and EPSR

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



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

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

See also:
- *Registers* on page B3-41.
- *APSR, Application Program Status Register* on page D1-880.
- *Interrupt Program Status Register (IPSR)*.
- *Execution Program Status Register (EPSR)* on page B3-46.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VWTF	From 8.0	None	-
XGTP	From 8.0	None	-

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.

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

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.

See also:

- *XPSR, APSR, IPSR, and EPSR* on page B3-45.
- *Function calls from Secure state to Non-secure state* on page B3-73.
- *IPSR, Interrupt Program Status Register* on page D1-1054.
- *BLX, BLXNS* on page C2-413.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DTBJ	From 8.0	None	Secure state requires S
XTCC	From 8.0	None	-
CDPK	From 8.0	None	-

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.

I_{GPJH} Bit[0] of the reset vector is 1 if the PE is to execute the code indicated by the reset vector.

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.

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

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KSCH	From 8.0	None	-
SQLX	From 8.0	None	A UsageFault requires M
LBJQ	From 8.0	None	-

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.

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`.

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 `NOP`s. That is, they behave as if they failed the Condition code check for the remaining iterations of the EPSR.IT state machine.

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 `NOP`s, as if they had failed the Condition code check.

I_{NJKF} The behavior might in some implementations vary from instruction to instruction, or between different instances of the same instruction.

R_{BWMN} Many instructions that are CONSTRAINED UNPREDICTABLE in an IT block are branch instructions or other non-sequential instructions that change the PC. Where these instructions are not treated as UNDEFINED within an IT block, the remaining iterations of the EPSR.IT state machine is treated in one of the following ways:

- EPSR.IT is cleared to 0.
- EPSR.IT advances for either a sequential or a nonsequential change of the PC in the same way as it does for instructions that are not CONSTRAINED UNPREDICTABLE that cause a sequential change of the PC.

I_{XHBL} This behavior does not apply to an instruction that is the last instruction in an IT block.

R_{TMWN} The instructions that are addressed by the updated PC do 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 `NOP`s. That is, they behave as if they had failed the Condition code check for the remaining iterations of the EPSR.IT state machine.

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`.

See also:

- *XPSR, APSR, IPSR, and EPSR* on page B3-45.
- *Execution Program Status Register (EPSR)* on page B3-46.

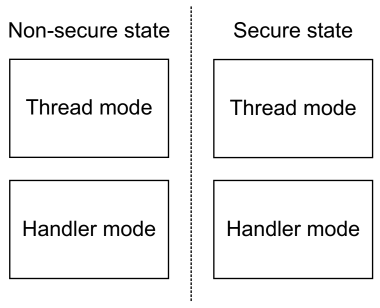
The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WWVX	From 8.0	None	Debug state requires Halting debug
CPDC	From 8.0	None	Debug state requires Halting debug
HVNS	From 8.0	None	Debug state requires Halting debug
LLDK	From 8.0	None	-
BWMN	From 8.0	None	-
TMWN	From 8.0	None	-
KVXD	From 8.0	None	-

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.



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.

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

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.

See also:

- *PE modes, Thread mode and Handler mode* on page B3-39.
- *Privileged and unprivileged execution* on page B3-40.
- *Security states, register banking between them* on page B3-50.
- *Security states, exception banking* on page B3-59.
- *Security state transitions* on page B3-71.
- *Resets, Cold reset, and Warm reset* on page B1-32.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HKKL	From 8.0	S	-
PBGT	From 8.0	S	-
HWFV	From 8.0	S	-
PLGH	From 8.0	!S	-

B3.7 Security states, register banking between them

IMGRQ 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.

RBHDK 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.

IGBWT 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:

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

IMKKR This specification uses the following naming convention to identify banked registers:

<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.

See also:

- *Registers* on page B3-41.
- *Security states, Secure state, and Non-secure state* on page B3-49.
- *Stack pointer* on page B3-51.
- *The System Control Space (SCS)* on page B6-192.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
BHDK	From 8.0	S	-

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

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

Stack	Stack pointer register
Main	MSP
Process	PSP

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

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

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

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

R_{XPWM} Bits [1:0] of the MSP or PSP, in either Security state, are always treated as RES0, so that all stack pointers are always guaranteed to be word-aligned.

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

- The value that is read or written from or to the SP is UNKNOWN.
- The instruction is permitted to be treated as UNDEFINED.
- If the SP is being written, it is UNKNOWN whether a stack-limit check is applied.

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

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

See also:

- *Security states, Secure state, and Non-secure state* on page B3-49.
- *PE modes, Thread mode and Handler mode* on page B3-39.
- *Exception entry, context stacking* on page B3-78.
- *Vector tables* on page B3-98.
- *Registers* on page B3-41.
- *Stack limit checks* on page B3-84.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RDLR	From 8.0	S	-
TGHV	From 8.0	None	-
TXRW	From 8.0	None	-
BTVD	From 8.0	!S	-
MDXK	From 8.0	S	-
XPWM	From 8.0	None	-
JXJM	From 8.0	None	-

B3.9 Exception numbers and exception priority numbers

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

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

Exception	Exception number	Priority number
Reset	1	-4[a]
Secure HardFault when AIRCR.BFHFNMINS is 1[b]	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[c]	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

a. Highest priority.

b. 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.

c. In a PE without the Security Extension, exception number 7 is reserved.

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[a]
Secure HardFault when AIRCR.BFHFNMINS is 1[b]	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

a. Highest priority.
b. 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.

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

R$_{\text{QQTT}}$ 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.

I$_{\text{QWTM}}$ 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).
- DebugMonitor exception.

I$_{\text{JQPH}}$ All other configurable exceptions can be configured using the NVIC_IPR.PRI_N<n> register fields.

R$_{\text{NFSM}}$ Configurable priority numbers start at 0, the highest configurable exception priority number.

R_{GGCP}

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

Number of priority bits of SHPRn.PRI_n implemented[a]	Number of configurable priority numbers	Minimum priority number (highest priority)	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

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

R_{CMGH}

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

Number of priority bits of SHPRn.PRI_n implemented[a]	Number of configurable priority numbers	Minimum priority number (highest priority)	Maximum priority number (lowest priority)
2	4	0	`0b11000000` = 192

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

See also:

- *Security states, exception banking* on page B3-59.
- *Faults* on page B3-62.
- *Priority model* on page B3-66.
- SHPR1, SHPR2, SHPR3.
- NVIC_IPRn.
- *ExecutionPriority* on page E2-1252.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CMTC	From 8.0	M	• A Secure HardFault and a SecureFault require S
MGNV	From 8.0	!M	• A Secure HardFault requires S • A SysTick exception requires ST
QQTT	From 8.0	None	-

Rule	Architecture version	Required extensions	Notes
NFSM	From 8.0	None	-
GGCP	From 8.0	M	-
CMGH	From 8.0	!M	-

B3.10 Exception enable, pending, and active bits

I_{QQDG} The SHCSR, ICSR, DEMCR, NVIC_IABRn, NVIC_ISPRn, NVIC_ISERn, and STIR contain exception enable, pending, and active fields.

IGHGW The following exceptions are always enabled and therefore do not have an exception enable bit:

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

ILHSX 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.

IQDKX 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.USERSETMPEND_S is 1, can use STIR to pend any Secure or Non-secure interrupt, otherwise a BusFault is generated.	When CCR.USERSETMPEND_NS is 1, can use STIR to pend a Non-secure interrupt, otherwise a BusFault is generated.

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

	Secure state	Non-secure state
Privileged execution	Can use STIR_NS to pend a Non-secure interrupt.	RES0
Unprivileged execution	When CCR.USERSETMPEND_NS is 1, can use STIR_NS to pend a Non-secure interrupt, otherwise 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	Fault	Fault

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

	Secure state	Non-secure state
Privileged execution	Can use NVIC_ISPRn_NS to pend a Non-secure interrupt.	RES0
Unprivileged execution	Fault	Fault

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

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

B3.11 Security states, exception banking

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

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

Exception	Banked
Reset	No
HardFault	Yes (conditionally)
NMI	No
MemManage fault[a]	Yes
BusFault[a]	No
UsageFault[a]	Yes
SecureFault[a]	No
SVCall	Yes
DebugMonitor[a]	No
PendSV	Yes
SysTick[b]	Yes
External interrupt 0	No
.	.
.	.
.	.
External interrupt N	No

a. Exception type is present only if the Main Extension is implemented.

b. This 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.

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, when 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 original exception that caused the vector fetch and not the current Security state.
- Faults triggered by the stacking of callee registers always target the Secure state.

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

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

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

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

R_{MQWN} SecureFault always targets Secure state.

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

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

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

<exception> is one of:
- NMI.
- BusFault.
- DebugMonitor.
- External interrupt N.

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

See also:
- *Exception numbers and exception priority numbers* on page B3-53.
- *Vector tables* on page B3-98.
- *SHCSR, System Handler Control and State Register* on page D1-1129.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PJHV	From 8.0	None	• A MemManage fault, BusFault, and a UsageFault exception require M • A SecureFault requires S • A Debug Monitor exception requires DebugMonitor exception • A SysTick exception requires ST
LNWV	From 8.0	S	• A UsageFault requires M • Floating-point state requires FP
GVPG	From 8.0	None	-
WLGH	From 8.0	S && ST	-
MQWN	From 8.0	S	-
PBXL	From 8.0	S	• A BusFault exception requires M • A DebugMonitor exception requires DebugMonitor exception

B3.12 Faults

I_{NHTB}

INHTB 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 and Security Extension are 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 MMFSR, BFSR, 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.

RXMRH MMFAR is updated only for a MemManage fault on a direct data access.

RDDJJ BFAR is updated only for a BusFault on a data access, a precise fault.

RFLDT 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

RTSCG 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.

RQPJS 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.

RGBJF 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.

ISCMW Arm strongly recommends that either BFAR is banked between Security states, or, if a single register is implemented, it is cleared when changing AIRCR.BFHFNMINS so as not to expose the last accessed address to the other Security state.

RKJPM In a PE with the Main Extension, the faults are:

Exception number	Exception	Fault	Fault status bit
3	HardFault	HardFault on a 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 a direct data access	MMFSR.DACCVIOL
		MemManage fault on context unstacking by hardware, because of an MPU access violation	MMFSR.MUNSTKERR
		MemManage fault on context stacking by hardware, because of an MPU access violation	MMFSR.MSTKERR
		When lazy Floating-point context preservation is active, a MemManage fault on saving Floating-point context to the stack	MMFSR.MLSPERR
5	BusFault	BusFault on an instruction fetch, precise	BFSR.IBUSERR
		BusFault on a data access, precise	BFSR.PRECISERR
		BusFault on a data access, imprecise	BFSR.IMPRECISERR
		BusFault on 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 when CCR.UNALIGN_TRP is 1	UFSR.UNALIGNED
		UsageFault, divide by zero when CCR.DIV_0_TRP is 1	UFSR.DIVBYZERO

Exception number	Exception	Fault	Fault status bit
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

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

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

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

Exception number	Exception
3	HardFault

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.

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

See also:
- *Exception numbers and exception priority numbers* on page B3-53.
- *Hardware-controlled priority escalation to HardFault* on page B3-100.
- Chapter B11 *Debug*.
- Chapter D1 *Register Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XMRH	From 8.0	M	-
DDJJ	From 8.0	M	-
FLDT	From 8.0	M	-
TSCG	From 8.0	M && !S	-
QPJS	From 8.0	M && S	-
GBJF	From 8.0	M && S	-

Rule	Architecture version	Required extensions	Notes
KJPM	From 8.0	M	A SecureFault requires S
WHBK	From 8.0	!M	
FQJV	From 8.0	S	-

B3.13 Priority model

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

Active

An exception that either:

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

Pending

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

Inactive

The exception has not been generated.

Active and pending

One instance of the exception is active, and a second instance of the exception is pending.

Only asynchronous exceptions can be active and pending. Synchronous exceptions are either inactive, pending, or active.

R_{CJDM} Lower priority numbers take precedence.

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_n and NVIC_IPRn values.
2. In a PE with or without the Main Extension applying the effects of PRIMASK.PM and AIRCR.PRIS.
3. In a PE with the Main Extension applying the effects of FAULTMASK.FM.
4. The execution priority is the either:
   - The exception with the lowest priority number.
   - The exception with the lowest priority group value.

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

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:

	BASEPRI, SHPRn.PRI_n [7:0], and NVIC_IPRn.PRI_Nn[7:0][a]	
AIRCR.PRIGROUP value	**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]

a. All low-order bits of these fields that are not implemented as priority bits are RES0.

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

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

a. SHPRn.PRI_n [5:0] are RES0.

RCQRV 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.

RWQWK 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 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.

INCDS The following is an example of exceptions with different priorities:

Example B3-1

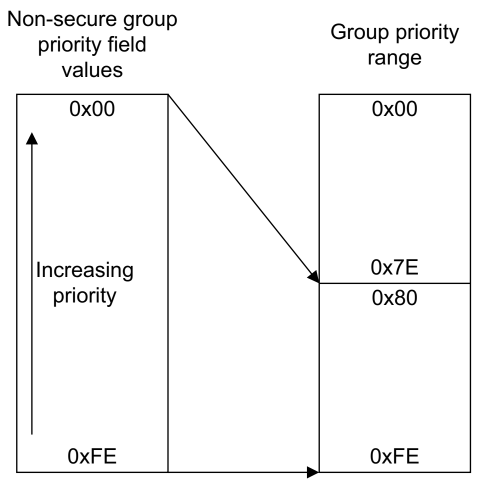
This example considers the following exceptions, that all have configurable priority numbers:

- 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.

IXFVH 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	0xFE

In this example, Secure exceptions in the range `0x00` - `0x7E` have priority over all Non-secure exceptions.

IWPCP 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 field value	**Mapped to**
0x00	0x80
0x40	0xA0
0x80	0xC0
0xC0	0xE0

See also:

- *Exception numbers and exception priority numbers* on page B3-53.

- *Special-purpose mask registers, PRIMASK, BASEPRI, FAULTMASK, for configurable priority boosting* on page B3-101.
- *Hardware-controlled priority escalation to HardFault* on page B3-100.
- *ExceptionPriority* on page E2-1244.
- *ExecutionPriority* on page E2-1252.
- *ComparePriorities* on page E2-1221.
- *RawExecutionPriority* on page E2-1320.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CJDM	From 8.0	None	-
HLJC	From 8.0	None	-
RKCQ	From 8.0	None	-
CQRV	From 8.0	None	A Secure exception requires S
WQWK	From 8.0	S	-

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.

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

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

R$_{XSPQ}$ For 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

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.

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

See also:
- *Security state transitions* on page B3-71.
- *DAP access permissions* on page B11-237

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CHJX	From 8.0	S	-
MSNJ	From 8.0	S	-
VHRL	From 8.0	S	-
XSPQ	From 8.0	S	-
TDNR	From 8.0	S	-

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 branch target address	Conditions for a change in Security state
Secure	Non-secure	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 Secure state if both: • The branch target address contains an `SG` instruction which is fetched and executed. • The whole of the instruction at the branch target address is flagged as Secure and Non-secure callable.

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.

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.

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 Secure fault or Secure HardFault

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 INVTRAN Secure fault or Secure HardFault

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 Secure fault or Secure HardFault is generated:

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

See also:

- *Instruction set, interstating support* on page C1-321.

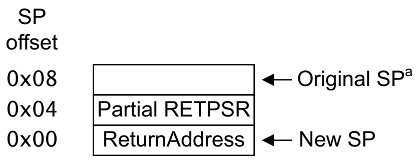
The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PQHT	From 8.0	S	-
WKXR	From 8.0	S	An INVTRAN SecureFault requires M
JKJD	From 8.0	S	An INVTRAN SecureFault requires M
XNVW	From 8.0	S	An INVTRAN SecureFault requires M
DWXH	From 8.0	S	-

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

RGVBB 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.

RQVJT Behavior is UNPREDICTABLE when a function call stack frame is not doubleword-aligned.

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

See also:

- *Instruction set, interstating support* on page C1-321.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GVBB	From 8.0	S	-
QVJT	From 8.0	S	-

B3.17 Function returns from Non-secure state

RHPFG An interstating function return begins when one of the following instructions loads a FNC_RETURN value into the PC:

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

On detecting a FNC_RETURN value in the PC:

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

RTFCK If the stack pointer is not 8 byte aligned the behavior is UNPREDICTABLE.

RDWTF 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 a FNC_RETURN value.

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

0	Secure state.
1	Non-secure state.

RQLJT 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.

RNTNW 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.

RFGNB 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.

RLNFB If the IPSR contains a value that is not supported by the PE the value is UNKNOWN and a INVPC UsageFault is generated.

RXMFG If the IPSR contains a value that is not supported by the PE the value is UNKNOWN and HardFault is generated.

IKBXQ 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.

See also:

- *Hardware-controlled priority escalation to HardFault* on page B3-100.
- *Lockup* on page B3-103.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HPFG	From 8.0	S	-
TFCK	From 8.0	S	-
DWTF	From 8.0	S	-
QLJT	From 8.0	M && S	-
NTNW	From 8.0	!M && S	-
FGNB	From 8.0	S	-
LNFB	From 8.0	M && S	-
XMFG	From 8.0	!M && S	-

B3.18 Exception handling

R_{YFHR}

An exception that does not cause *lockup* sets both:

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

R_{VLDB}

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

R_{WBND}

Preemption of current execution causes the following basic sequence:

1. R0-R3, R12, LR, RETPSR, and 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.
    - Lazy Floating-point state preservation.
4. LR is set to EXC_RETURN.
5. Optional clearing of 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. A transition to the Security state of the exception being activated.
8. 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.
9. CONTROL.SPSEL is set to 0, to indicate the selection of the main stack, dependent on the Security state being targeted.
10. 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.
11. The Security state is changed to the Security state of the exception that is being activated.
12. 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.
13. EPSR.T is set to bit[0] of the exception vector for the exception to be taken.
14. The PC is set to the exception vector for the exception to be taken.

I_{PSGQ}

The `HandleException()`, `ExceptionEntry()`, `PushStack()`, `ExceptionTaken()`, and `ActivateException()` pseudocode describes the full exception handling sequence.

R_{NJVF}

During exception entry, if it is found that the exception and the exception vector are associated with different Security states, an INVEP or INVTRAN SecureFault is generated, unless 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.

R_{QLHB}

The return address is one of the following:

- On return from a synchronous exception, other than an SVCall exception, the address of the instruction that caused the exception.
- On return from an asynchronous exception, the address of the next instruction in the program order.
- On return from an SVCall exception, the address of the next instruction in the program order.

R_{XKDD}

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

See also:

- *Exception enable, pending, and active bits* on page B3-57.
- *Priority model* on page B3-66.
- *Exception entry, context stacking* on page B3-78.
- *Exception entry, register clearing after context stacking* on page B3-83.
- *Vector tables* on page B3-98.
- *Stack limit checks* on page B3-84.
- *Exceptions during exception entry* on page B3-91.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
YFHR	From 8.0	None	-
VLDB	From 8.0	None	-
WBND	From 8.0	None	Some steps might require additional extensions
NJVF	From 8.0	S	An INVEP or INVTRAN SecureFault requires M
QLHB	From 8.0	None	-
XKDD	From 8.0	None	-

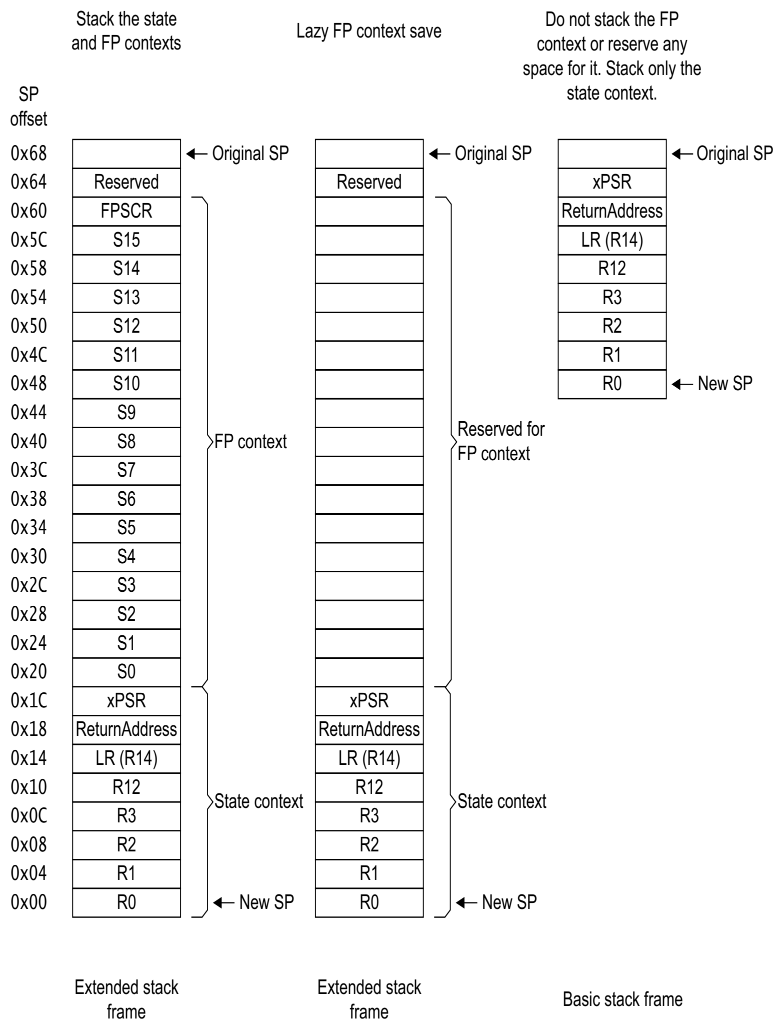
B3.19 Exception entry, context stacking

RPWWG 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.

RPTRL 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*:

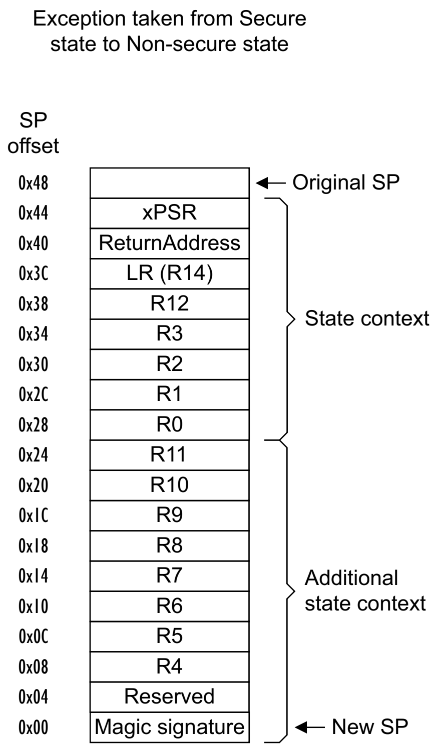
- Stack the Floating-point context.
- Reserve space on the stack for the Floating-point context. This is called *lazy Floating-point context preservation*.



RPLHM 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 is to Non-secure state, regardless of whether a higher priority derived or late-arriving exception targeting Secure state occurs, the PE hardware extends the stack frame, and also saves *additional state context*, as shown here:



R$_{\text{DHPD}}$ 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.

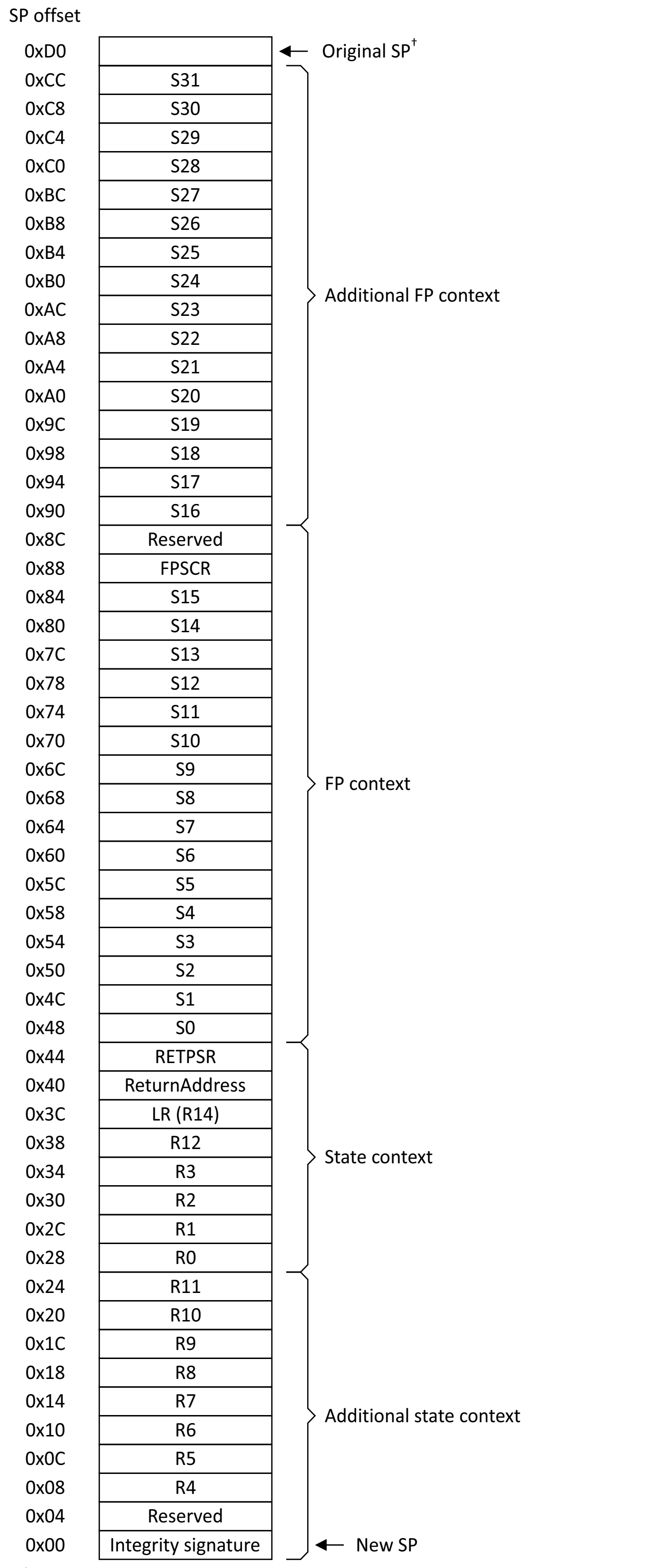
Secure state when CONTROL.FPCA is 1

The PE hardware:

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

   If FPCCR_S.TS is 1 when the exception is taken, the PE hardware either stacks both the Floating-point context and additional Floating-point context, or 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 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:



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

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].

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

1	The PE realigned the stack.
0	The PE did not realign the stack.

R_{QDKQ} Full descending stacks are used.

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

- Because setting FPCCR.ASPEN to one causes the PE to automatically set CONTROL.FPCA to 1 on the execution of a floating-point instruction, setting FPCCR.ASPEN to one 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 which of the above the PE hardware performs:

0 The PE hardware automatically stacks FP context on taking an exception. In a PE that also includes the Security Extension, if FPCCR_S.TS == 1, 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, 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.

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

R_{MBXL} 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, as stored in the 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. Sets FPCCR.LSPACT to 0 to indicate that lazy Floating-point context preservation is no longer active.
  3. 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.

R_{LGNS} When the following conditions are met on exception entry, the PE generates a Secure NOCP UsageFault and does not allocate space on the stack for Floating-point context:

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

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

R_{KMBN} If lazy Floating-point context preservation is activated when FPCCR.LSPACT is already set to 1, the PE generates an LSERR SecureFault.

RFVTL 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.

See also:

- *Stack pointer* on page B3-51.
- *Exception entry, register clearing after context stacking* on page B3-83.
- *Integrity signature* on page B3-90.
- *PushStack* on page E2-1316.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PWWG	From 8.0	None	-
PTRL	From 8.0	!S && FP	-
PLHM	From 8.0	S	-
DHPD	From 8.0	S && FP	-
BKVD	From 8.0	None	-
QDKQ	From 8.0	None	-
PWBW	From 8.0	FP	Space is reserved for both the FP context and additional FP context if S is implemented
GHDJ	From 8.0	FP	-
MBXL	From 8.0	FP	-
LGNS	From 8.0	S && FP	-
QLGM	From 8.0	None	-
KMBN	From 8.0	S && FP	-
FVTL	From 8.0	S && FP	-

B3.20 Exception entry, register clearing after context stacking

R_{DRRB} The PE hardware sets R0-R3, R12, APSR, and EPSR to an UNKNOWN value after it has pushed *state context* to the stack.

R_{DJRX} In a PE:

- 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.

R_{SNDB} After the PE hardware has pushed state context to the stack, it sets R0-R3, R12, APSR, and EPSR to:

- An UNKNOWN value if the exception is taken to Secure state.
- Zero if the exception is taken to Non-secure state.

If the PE did not also push *additional state context* to the stack, as indicated by EXC_RETURN.DCRS, the values of R4-R11 remain unchanged.

If the PE also pushed additional state context to the stack, as indicated by EXC_RETURN.DCRS, then afterwards:

- If the Background state is Non-secure, R4-R11 remain unchanged.
- If the Background state is Secure, the PE sets R4-R11 to:
  - — An UNKNOWN value if the exception is taken to Secure state.
  - — Zero if the exception is taken to Non-secure state.

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

State context and additional state context

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

Floating-point context and additional Floating-point context

- 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.

In both cases, CONTROL.FPCA is set to 0 to indicate that the Floating-point Extension is not active.

See also:

- *Exception entry, context stacking* on page B3-78.
- *Tail-chaining* on page B3-93.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DRRB	From 8.0	None	-
DJRX	From 8.0	FP	-
SNDB	From 8.0	S	-
JWBK	From 8.0	S && FP	-

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.

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 register
Secure	Main	MSPLIM_S
	Process	PSPLIM_S

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

Stack	Stack limit register
Main	MSPLIM
Process	PSPLIM

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

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

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

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

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

- Exception entry.
- Tail-chaining from a Secure to a Non-secure exception.
- A function call from Secure code to Non-secure code.

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.

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.

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

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

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

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

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

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

I_{BJHX} When an instruction updates the stack pointer, if it results in a violation of the stack limit, it is the modification of the stack pointer that generates the exception, rather than an access that uses the out-of-range stack pointer.

R_{DBSG} On a violation of a stack limit when an instruction updates the stack pointer:

- It is IMPLEMENTATION DEFINED whether accesses to addresses equal to or above the stack limit value are performed.
- It is IMPLEMENTATION DEFINED whether the destination register or registers of load instructions are updated as long as the base register, stack pointer, and PC are not modified.
- Accesses below the stack limit are not performed.

I_{RRDX} CCR.STKOFHFNMIGN controls whether stack limit violations are IGNORED while executing at a requested execution priority that is negative.

R_{XCQL} It is UNKNOWN whether a stack limit check is performed on any use of the SP marked as UNPREDICTABLE.

R_{JXFP} Store operations using the SP as a base register do not perform any stores below the associated stack limit address.

R_{JSLC} It is UNKNOWN whether Load/Store instructions that specify the SP as a base register and attempt a read or write below the associated stack limit but write-back a value greater than the stack limit address generate an STKOF UsageFault.

See also:

- *Stack pointer* on page B3-51.
- *Tail-chaining* on page B3-93.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PCRT	From 8.0	!M && !S	-
NHBX	From 8.0	!M && S	-
JPFX	From 8.0	M && !S	-
XQDS	From 8.0	M && S	-
TCXN	From 8.0	None	-
DKSR	From 8.0	None	Secure exceptions and secure code require S
ZLZG	From 8.0	None	A UsageFault requires M
CCSC	From 8.0	S	A UsageFault requires M
GGRH	From 8.0	None	-
YVWT	From 8.0	None	-
DBSG	From 8.0	None	-
XCQL	From 8.0	None	-
JXFP	From 8.0	None	-
JSLC	From 8.0	M	-

B3.22 Exception return

R_KPSS The PE begins an exception return when both of the following are true:

- The PE is in Handler mode.
- One of the following instructions loads an EXC_RETURN value, `0xFFXXXXXX`, into the PC:
  - A `POP (multiple registers)` or `LDM` that includes loading the PC.
  - An `LDR` with the PC as a destination.
  - A `BX` with any register.
  - A `BXNS` with any register.

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

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

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

R_TXDW Behavior is UNPREDICTABLE if EXC_RETURN.FType is 0 and the Floating-point Extension register file is not implemented.

R_TNSK Behavior is UNPREDICTABLE if EXC_RETURN[23:7] are not all 1 and if bit[1] is not 0.

R_XLCP Behavior is UNPREDICTABLE if any of the following are true and the Security Extension is not implemented:

- EXC_RETURN.S is 1.
- EXC_RETURN.DCRS is 0.
- EXC_RETURN.ES is 1.

R_LLBT The following integrity checks on exception return are performed on every exception return:

1. In a PE with the Security Extension, the integrity check that is called the *EXC_RETURN.ES validation check*, as follows:
   - If the PE was in Non-secure state when EXC_RETURN was loaded into the PC and either EXC_RETURN.DCRS is 0 or EXC_RETURN.ES is 1, an INVER SecureFault is generated and the PE sets EXC_RETURN.ES to 0.
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. If the PE includes the Security Extension, the HardFault targets the Security state that EXC_RETURN.S specifies.
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. If the PE includes the Security Extension, the HardFault targets the Security state that EXC_RETURN.S specifies.

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*.

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.

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.

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.

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.

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.

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, or remains unchanged if it was 0.

On exception return, the inverse of EXC_RETURN.FType is written to CONTROL.FPCA.

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.

I_{DDWR} Attempts to access NSACR when the Floating-point unit is disabled result in a NOCP UsageFault.

I_{NQNR} `IsCPEnabled()` indicates the prioritization if the access is blocked by multiple registers.

R_{XNNG} When the following conditions are met on exception return, the PE generates an LSERR SecureFault:
- EXC_RETURN.FType is 0.
- The stack might contain Secure Floating-point context, that would be unstacked on the return. That is, FPCCR_S.LSPACT is 1.
- The return is to Non-secure state.

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.

R_{GDVT} The floating-point registers are not modified if the checks prior to unstacking fail.

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.

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.

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.

See also:
- *Exception handling* on page B3-76.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KPSS	From 8.0	None	-
TXDW	From 8.0	None	-
TNSK	From 8.0	None	-
XLCP	From 8.0	None	-
LLBT	From 8.0	None	Some steps require additional extensions, as listed in the rule
HXSR	From 8.0	S	-
DQLL	From 8.0	S	-
JMJC	From 8.0	S	-
RPGL	From 8.0	S	-
CGML	From 8.0	FP	A SecureFault requires S
XNNG	From 8.0	S && FP	-
VGGF	From 8.0	FP	-
GDVT	From 8.0	FP	-
HNNW	From 8.0	S && FP	-
LMNG	From 8.0	!S && FP	-
HRJH	From 8.0	None	-

B3.23 Integrity signature

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

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

Stack Frame Type Check

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

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

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

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

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.

See also:

- *Exception entry, context stacking* on page B3-78.
- *Exception return* on page B3-87.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PHBP	From 8.0	S &&FP	-
MVKS	From 8.0	!FP && S	A SecureFault requires M

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. In this case:
  - — 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.

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

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

R_{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.

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.DCRS is set to 0.

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

See also:

- *Exception numbers and exception priority numbers* on page B3-53.
- *Priority model* on page B3-66.
- *Exception handling* on page B3-76.
- *Tail-chaining* on page B3-93.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MRTR	From 8.0	None	-
BXTB	From 8.0	None	-
GDNT	From 8.0	None	A DebugMonitor exception requires DebugMonitor exception
GVHV	From 8.0	S	-

B3.25 Exceptions during exception return

I_{KXPV} During exception return exceptions can occur, for example asynchronous exceptions, or the exception return might itself cause an exception.

Any exception that occurs during exception return is a late-arriving exception.

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

R_{TRFM} When a late-arriving exception during exception return is higher priority than the priority being returned to, the PE takes the late-arriving exception by using tail-chaining.

I_{MBNG} The architecture does not specify the point during exception return at which the PE recognizes the arrival of an asynchronous exception. If a PE recognizes an asynchronous exception after an exception return has completed, there is no opportunity to tail-chain the asynchronous exception.

R_{MJDN} If the priority of a derived exception during exception return is equal to or lower 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.

R_{DHFK} If the priority of a derived exception during exception return, after priority escalation if appropriate, is higher priority than the priority being returned to, the PE uses tail-chaining to take the derived exception.

See also:

- *Exception numbers and exception priority numbers* on page B3-53.
- *Priority model* on page B3-66.
- *Exception return* on page B3-87.
- *Tail-chaining* on page B3-93.
- *DebugMonitor exception* on page B11-245.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
TRFM	From 8.0	None	-
MJDN	From 8.0	None	A DebugMonitor exception requires DebugMonitor exception
DHFK	From 8.0	None	-

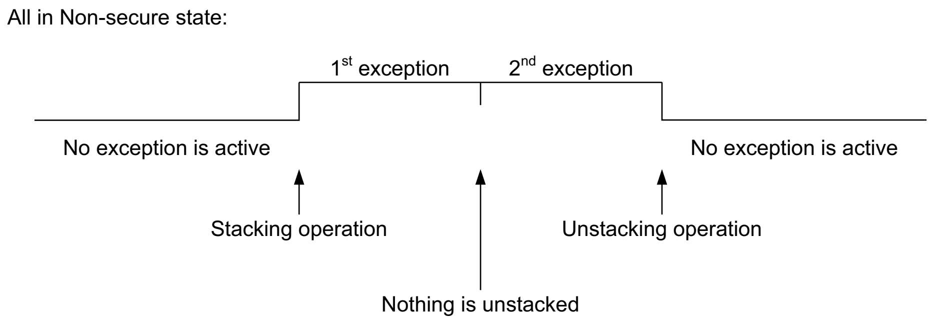
B3.26 Tail-chaining

RFKXX *Tail-chaining* behavior is as follows:

On detecting an EXC_RETURN value in the PC, if there is a pending exception that is higher priority than the execution priority being returned to, the PE hardware:

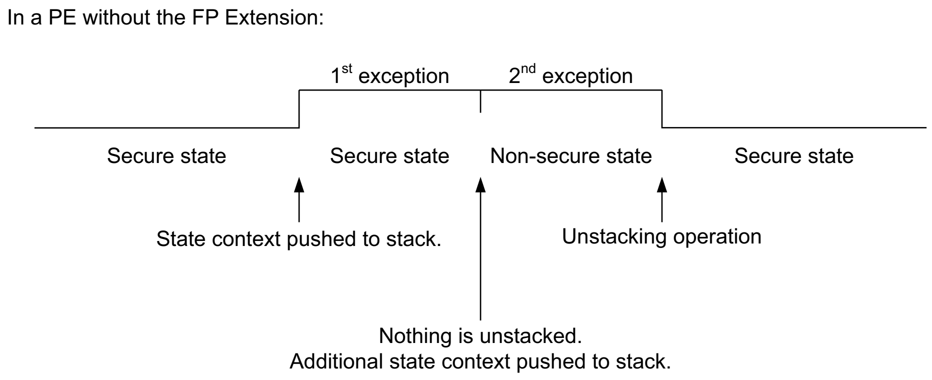
1. Does not unstack the stack.
2. Takes the pending exception.
    - The PE will tail-chain any derived exception on exception return if the derived exception has higher priority than the execution priority being returned to.
    - The PE will tail-chain any synchronous fault on exception return if the 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 thread of execution that has the priority that would have been returned to but for the tail-chained exception.

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



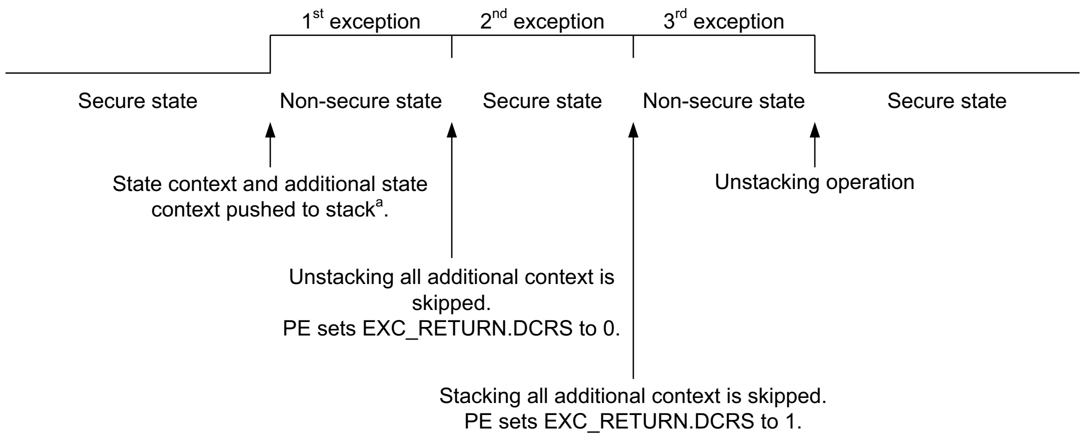
IRWDT If tail-chaining prevents a derived exception on exception return, the derived exception might instead be generated on the return from the last tail-chained exception.

RPXVB 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:



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:


1st exception
2nd exception
3rd exception
Secure state
Non-secure state
Secure state
Non-secure state
Secure state
State context and additional state context pushed to stack[a].
Unstacking operation
Unstacking all additional context is skipped.
PE sets EXC_RETURN.DCRS to 0.
Stacking all additional context is skipped.
PE sets EXC_RETURN.DCRS to 1.

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

I_{TMVF}

When multiple exceptions are tail-chained, 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.

I_{CVRD}

Arm recommends that Secure exception handlers clear the Floating-point context registers to zero before they return.

I_{LNPQ}

If FPCCR.CLRONRET is set to 1, hardware automatically clears the Floating-point context registers to zero on exception return.

R_{JMHS}

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

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

See also:

- *Exception entry, context stacking* on page B3-78.
- *Exceptions during exception return* on page B3-92.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FKXX	From 8.0	None	-
PXVB	From 8.0	S	-
JMHS	From 8.0	None	-

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*.

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*.

R_{KCMD} To support *instruction restart*, singleword load instructions do not update the destination register when the PE takes an exception during execution.

I_{NDQT} Instructions that the PE can halt to use instruction resume are called *exception-continuable instructions*.

R_{LGPQ} The exception-continuable instructions are `LDM`, `LDMDB`, `STM`, `STMDB`, `POP (multiple registers)`, and `PUSH (multiple registers)`.

R_{RDHK} In a PE with the Floating-point Extension, the floating-point exception-continuable instructions are `VLDM`, `VLLDM`, `VLSTM`, `VSTM`, `VPOP`, and `VPUSH`.

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.

R_{QWWW} It is IMPLEMENTATION DEFINED whether a `VLLDM` or `VLSTM` instruction aborts or completes when an interrupt occurs.

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 exception-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`

R_{XFGN} Behavior is UNPREDICTABLE if EPSR.IT/ICI contains a valid EPSR.IT/ICI non-zero value and the register number that it contains is either:

- Not in the register list of the exception-continuable Load Multiple or Store Multiple instruction.
- The first register in the register list of the exception-continuable Load Multiple or Store Multiple instruction.

R_{LRGK} The PE generates an INVSTATE UsageFault if EPSR.IT/ICI contains a valid nonzero value and the instruction being executed is not a Load Multiple or Store Multiple instruction. A fault is not generated if the instruction is a BKPT instruction.

R_{JXKQ} If the PE uses instruction resume during a Load 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.IT/ICI as already loaded.
- The base register.
- The PC.

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

R

$\mathrm{R}_{\mathrm{NKNQ}}$ 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.

$\mathrm{R}_{\mathrm{LSCQ}}$ 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.

$\mathrm{R}_{\mathrm{RFGF}}$ 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.IT/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.IT/ICI is cleared to zero.

$\mathrm{R}_{\mathrm{SGWB}}$ When a Load Multiple instruction includes its base register in its register list, if the PE takes an exception during the instruction:

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PGRC	From 8.0	M	-
KRLL	From 8.0	None	-
KCMD	From 8.0	None	-
LGPQ	From 8.0	M	-
RDHK	From 8.0	FP	-
VFBX	From 8.0	S && FP	-

Rule	Architecture version	Required extensions	Notes
QWWW	From 8.0	M	-
QVFC	From 8.0	None	Some instructions listed require FP
XFGN	From 8.0	None	-
LRGK	From 8.0	M	-
JXKQ	From 8.0	M	-
NKNQ	From 8.0	None	Instruction resume requires M
LSCQ	From 8.0	!M	-
RFGF	From 8.0	M	-
SGWB	From 8.0	None	-

B3.28 Vector tables

R_{NWFF}

RNWFF 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 RAZ/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.

RGTJQ 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 RAZ/WI.

IWFGX Arm recommends that VTOR_S points to memory that is Secure and not Non-secure callable.

RWPRT 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

RLFDL 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).

RXPPT 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.

IBVSC 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.

RVDPD Vector fetches for entries beyond the natural alignment of the associated VTOR occur from an UNKNOWN entry within the vector table.

I~PLSB~ Arm recommends that it is ensured that the vector table and VTOR are aligned so that the entry for the highest taken exception falls within the natural alignment of the table, and at a minimum that the vector table is 128 byte aligned. A PE might impose further restrictions on the VTOR.

R~HBSS~ If a vector fetch causes a Security attribution unit violation or an IMPLEMENTATION DEFINED attribution unit violation, 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.
- If the original exception and the VECTTBL HardFault are different, or target different Security states, the VECTTBL HardFault becomes pending.

See also:
- *IMPLEMENTATION DEFINED Attribution Unit (IDAU)* on page B8-214.
- *Security attribution unit (SAU)* on page B8-212.
- *Exception numbers and exception priority numbers* on page B3-53.
- *Execution Program Status Register (EPSR)* on page B3-46.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
NWFF	From 8.0	S	-
GTJQ	From 8.0	!S	-
WPRT	From 8.0	None	-
LFDL	From 8.0	None	-
XPPT	From 8.0	M	-
VDPD	From 8.0	None	-
HBSS	From 8.0	S	-

B3.29 Hardware-controlled priority escalation to HardFault

R_{GNVS} When current execution has a priority number ≥0:

- If a synchronous exception with an equal or lower priority 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.

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

R_{PBJQ} When current execution has a priority number ≥ 0, 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 interrupt, the PE does not escalate the interrupt. The interrupt remains pending.

R_{DQRR} A fault that has been escalated to a HardFault retains the return address behavior of the original fault.

See also:

- *Exception numbers and exception priority numbers* on page B3-53.
- *DebugMonitor exception* on page B11-245.
- *Lockup* on page B3-103.
- *Security states, exception banking* on page B3-59

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GNVS	From 8.0	None	A DebugMonitor exception requires DebugMonitor exception
HPLM	From 8.0	None	-
PBJQ	From 8.0	None	-
DQRR	From 8.0	None	-

B3.30 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.

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.

R_{FHMC} The PRIMASK, FAULTMASK, and BASEPRI priority boosting mechanisms only boost the group priority, not the subpriority.

R_{SKBJ}

Without the Security Extension:

- An exception return other than from an NMI sets FAULTMASK to 0.

R_{HRTM}

With the Security Extension:

- An exception return other than from an NMI 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.

I_{PLKD}

The *raw execution priority* is:

- The execution priority minus the effects of AIRCR.PRIS == 1, and minus any configurable PRIMASK, FAULTMASK, or BASEPRI priority boosting.

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 .

See also:

- *Priority model* on page B3-66.
- *Exception numbers and exception priority numbers* on page B3-53.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FHMC	From 8.0	None	FAULTMASK and BASEPRI require M
SKBJ	From 8.0	!S	-
HRTM	From 8.0	S	-

B3.31 Lockup

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

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

R_{MBTM} When the PE is in lockup:

- DHCSR.S_LOCKUP reads as 1.
- The PC reads as `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.

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.

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

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

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.

See also:

- *Priority model* on page B3-66
- *Instruction execution* on page B3-104
- *Floating-point lazy Floating-point context preservation* on page B3-105.
- *Vector or stack pointer error on reset* on page B3-105.
- *Errors on preemption and stacking for exception entry* on page B3-106.
- *Vector read error on NMI or HardFault entry* on page B3-107.
- *Integrity checks on exception return* on page B3-108.
- *Errors when unstacking state on exception return* on page B3-108.
- Chapter B11 *Debug*

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MBTM	From 8.0	None	-
JRJC	From 8.0	None	Entry to Debug state requires Halting debug
HJNP	From 8.0	None	-
SPPN	From 8.0	None	Entry to Debug state requires Halting debug

B3.31.1 Instruction-related lockup behavior

Instruction execution

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

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

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.

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.

R_{DWKP} Asynchronous BusFaults do not cause lockup.

R_{KTQM} When a BusFault does not cause lockup, the value that is read or written to the location that generated the BusFault is UNKNOWN.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VGMR	From 8.0	M && S	
QMMB	From 8.0	!S	-
VGNW	From 8.0	None	-
DWKP	From 8.0	None	-
KTQM	From 8.0	None	-

Floating-point lazy Floating-point context preservation

R_{RNKB} When FPCCR.LSPACT is 1, a NOCP UsageFault, AU violation, MPU violation, or synchronous bus error 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.

R_{MMBJ} When FPCCR.LSPEN is 0, any faults that are caused by floating-point register reads or writes during exception entry or exception return are handled as faults on stacking or unstacking respectively.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RNKB	From 8.0	FP && S	• An MPU violation requires MPU
MMBJ	From 8.0	FP	-

B3.31.2 Exception-related lockup behavior

Vector or stack pointer error on reset

R_{BHVG} On reset, if reading the vector table to obtain either the vector for the reset handler or the initialization value for the main stack pointer causes a bus error, 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`.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BHVG	From 8.0	None	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 bus error 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.

R_{QSSB} When an AU violation, MPU violation, NOCP UsageFault, STKOF UsageFault, LSERR SecureFault, or synchronous bus error occurs during context stacking, it is IMPLEMENTATION DEFINED whether the PE continues to stack any of the remaining context.

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

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

At the point of lockup:

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VKTX	From 8.0	S	• An AU violation requires S • An MPU violation requires MPU • A UsageFault requires M • A SecureFault requires S
QSSB	From 8.0	None	• An AU violation requires S • An MPU violation requires MPU • A UsageFault requires M • A SecureFault requires S
GJJG	From 8.0	None	• An AU violation requires S • An MPU violation requires MPU • A UsageFault requires M • A SecureFault requires S

Vector read error on NMI or HardFault entry

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

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

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CTKP	From 8.0	None	-

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.

R_{WBVC}

The target Security state of an INVPC UsageFault generated because of a failed integrity check on exception return is either the Security state the exception return was executed in or the Background state dependent on when the INVPC UsageFault was generated.

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.
- Sets the PC to `0xEFFFFFFE`.

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
TRFJ	From 8.0	S	-
WBVC	From 8.0	M && S	-
DFKP	From 8.0	None	-

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. An AU violation, MPU violation, or synchronous bus error 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.

R_{XFCQ} When an AU violation, MPU violation, or synchronous bus error 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.FType.
- Sets the PC to `0xEFFFFFFE`.

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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
WKSJ	From 8.0	S	• An MPU violation requires MPU
XFCQ	From 8.0	None	• An AU violation requires S • An MPU violation requires MPU

B3.32 Context Synchronization Event

R_{QXWD} The architecture requires a Context synchronization event to guarantee visibility of any change to any memory-mapped register described in the architecture. Following a Context synchronization event a completed write to a memory-mapped register is visible to an indirect read by an instruction appearing in program order after the context synchronization event.

R_{TVHX} Between any change to a memory-mapped register and a subsequent Context synchronization event, it is UNPREDICTABLE whether an indirect read of the register by the PE uses the old or new values.

R_{RMMM} Where multiple changes are made to memory-mapped registers before a Context synchronization event, each value might independently be the old or new value.

R_{NSLQ} Where unsynchronized values apply to different areas of architectural functionality, or IMPLEMENTATION DEFINED functionality, those areas might independently treat the values as being either the old or new value.

R_{BKSX} The choice between the behaviors is IMPLEMENTATION DEFINED and might vary for each use of the unsynchronized value.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QXWD	From 8.0	None	-
TVHX	From 8.0	None	-
RMMM	From 8.0	None	-
NSLQ	From 8.0	None	-
BKSX	From 8.0	None	-

B3.33 Coprocessor support

RBSLX Coprocessor support is OPTIONAL.

I JBMG When coprocessors are not supported, the fields in CPACR, NSACR, and CPPWR that are associated with the unsupported coprocessor are RAZ/WI.

R XSQH The architecture supports 0-16 coprocessors, CP0 to CP15.

R HJDH CP0 to CP7 are IMPLEMENTATION DEFINED.

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.

R QSRC Arm reserves CP8 to CP15.

R LKZM CP10 to CP11 are reserved to support the Floating-point Extension, and CP10 controls the CP11 Floating-point instructions.

R LPMK The state that is associated with Floating-point unit described in CPPWR.SU10 applies to S registers, D registers, and FPSCR.

R XXDG Instructions that are issued to unimplemented or disabled coprocessors result in a NOCP UsageFault.

R RMLV If a coprocessor cannot complete an instruction, an UNDEFINSTR UsageFault is generated.

See also:
- Chapter B4 *Floating-point Support*.
- *CPACR, Coprocessor Access Control Register* on page D1-898.
- *CPPWR, Coprocessor Power Control Register* on page D1-900.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BSLX	From 8.0	M	-
XSQH	From 8.0	M	-
HJDH	From 8.0	M	-
XPRQ	From 8.0	M	Secure state requires S
QSRC	From 8.0	M	-
LKZM	From 8.0	FP	-
LPMK	From 8.0	FP	-
XXDG	From 8.0	M	-
RMLV	From 8.0	M	-

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}

I$_{VBNH}$ The optional Floating-point Extension defines a *Floating Point Unit* (FPU). Coprocessors 10 and 11 support the Extension.

I$_{RXQX}$ Floating-point is sometimes abbreviated to FP.

R$_{GQBM}$ The version of Floating-point Extension that is supported is FPv5.

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.
- A Default NaN 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).

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

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.

I$_{JDJQ}$ To use the Floating-point Extension, software must enable access to CP10, by programming CPACR.CP10.

R$_{PDMV}$ The value of CPACR.CP11 is UNKNOWN if it is not programmed to the same value as CPACR.CP10.

See also:

- *Armv8-M variants* on page A1-27.
- *The System Control Space (SCS)* on page B6-192.
- *About the Floating-point Status and Control Registers* on page B4-116.
- *Registers for floating-point data processing, S0-S31, or D0-D15* on page B4-117.
- *The Flush-to-zero mode* on page B4-124.
- *The Default NaN mode, and NaN handling* on page B4-126.
- *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page B4-120.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GQBM	From 8.0	FP	-
PDMV	From 8.0	FP	-

B4.2 About the Floating-point Status and Control Registers

R_{HCJS} The register map of the coprocessor System register space is as follows.

Location	Register	Information
`0b0001`	FPSCR.{N, Z, C, V}	Access to flags

All locations that are not explicitly listed in this table are reserved, and accesses to these locations result in UNPREDICTABLE behavior.

I_{GJWP} Software can use `VMRS` and `VMSR` instructions to access the Floating-point Status and Control registers.

R_{FXBJ} Execution of floating-point instructions that generate floating-point exceptions update the appropriate status fields of FPSCR.

See also:
- *Coprocessor support* on page B3-111.
- *The optional Floating-point Extension, FPv5* on page B4-114.
- *FPSCR, Floating-point Status and Control Register* on page D1-999.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

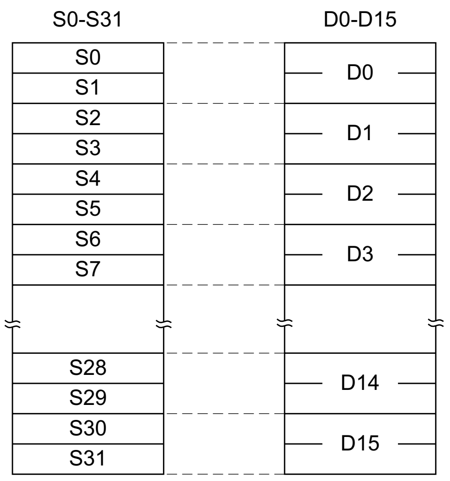
Rule	Architecture version	Required extensions	Notes
HCJS	From 8.0	FP	-
FXBJ	From 8.0	FP	-

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:



R_{XWJQ} After a reset, the values of S0-S31 or D0-D15 are UNKNOWN.

See also:

- *The optional Floating-point Extension, FPv5* on page B4-114.
- *Exception handling* on page B3-76.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
TWCB	From 8.0	FP	-
XWJQ	From 8.0	FP	-

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.

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

I_{BGPN} The following is called *Arm standard floating-point operation*:

- IEEE 754-2008 plus the following FPSCR configuration:
  — Flush-to-zero mode enabled.
  — Default NaN mode enabled.
  — Round to Nearest mode selected.
  — Alternative half-precision interpretation not selected.

See also:

- *IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008*.
- *The Flush-to-zero mode* on page B4-124.
- *The Default NaN mode, and NaN handling* on page B4-126.
- *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page B4-120.

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, +∞ and -∞.
- NaNs, signaling NaN and quiet NaN.

See also:

- *Floating-point standards and terminology* on page B4-118.
- *IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008*.
- *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page B4-120.

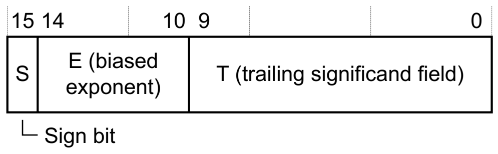
The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FWXC	From 8.0	FP	-

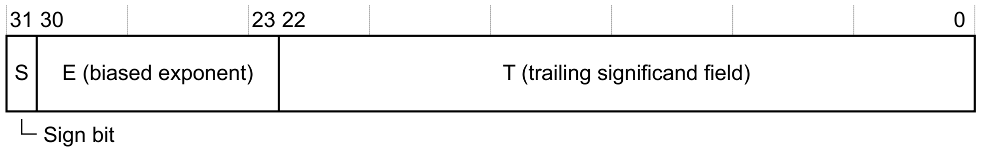
B4.6 Floating-point encoding formats, half-precision, single-precision, and double-precision

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.

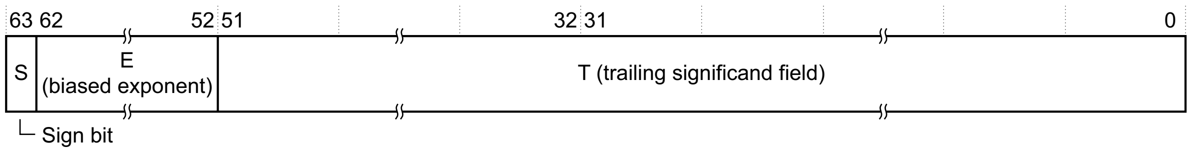
I~LGTJ~ The half-precision encoding format is:



I~CWBP~ The single-precision encoding format is:



I~FVWV~ The double-precision encoding format is:



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)	MSB of T[a]	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. Zero < E < `0xFF`, if single-precision format. Zero < E < `0x7FF`, if double-precision format.	-	-	-	A normalized number
`0x1F`, if half-precision format, IEEE interpretation. `0xFF`, if single-precision format. `0x7FF`, if double-precision format.	Non-zero	-	0	A signaling NaN
		-	1	A quiet NaN
	Zero	0	-	Infinity, +∞
		1	-	Infinity, -∞
`0x1F`, if half-precision, alternative half-precision interpretation.	-	-	-	A normalized number

a. MSB = most significant bit.

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

$(-1)^S \times 2^{-1022} \times (0.T)$

R_{PKXD} Denormalized numbers can be flushed to zero. FPv5 provides a Flush-to-zero mode.

See also:
- *IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008*.
- *Floating-point data representable* on page B4-119.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RHKS	From 8.0	FP	-
RWRW	From 8.0	FP	-
DPHH	From 8.0	FP	-
PKXD	From 8.0	FP	-

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.

I_{JCWS} The criteria for the Underflow exception to be generated are different in Flush-to-zero mode.

I_{NFHK} The corresponding status flags for the IEEE 754 floating-point exceptions are FPSCR.{IOC, DZC, OFC, UFC, IXC}.

See also:

- *IEEE 754-2008, IEEE Standard for Floating-point Arithmetic, August 2008*.
- *The Flush-to-zero mode* on page B4-124.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BCCL	From 8.0	FP	-

B4.8 The Flush-to-zero mode

I_{XGFP} Software can enable Flush-to-zero mode by setting FPSCR.FZ to 1.

I_{WMKJ} Using Flush-to-zero mode is a deviation from IEEE 754.

R_{JQHX} Half-precision floating-point numbers are exempt from Flush-to-zero mode.

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.

R_{KBJJ} When an input to a floating-point operation is flushed to zero, the PE generates an Input Denormal exception.

R_{SBCK} Input Denormal exceptions are only generated in Flush-to-zero mode.

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.

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 < `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.

R_{QPQF} When the result of a floating-point operation is flushed to zero, the PE generates an Underflow exception.

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.

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.

R_{RWRT} The PE does not generate an Inexact exception when a floating-point number is flushed to zero.

I_{SQCJ} The corresponding status flag for the Input Denormal exception is FPSCR.IDC.

See also:

- *The IEEE 754 floating-point exceptions* on page B4-123.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JQHX	From 8.0	FP	-
VJSF	From 8.0	FP	-
KBJJ	From 8.0	FP	-
SBCK	From 8.0	FP	-
WJDM	From 8.0	FP	-

Rule	Architecture version	Required extensions	Notes
PHPT	From 8.0	FP	-
QPQF	From 8.0	FP	-
TPVD	From 8.0	FP	-
RTPH	From 8.0	FP	-
RWRT	From 8.0	FP	-

B4.9 The Default NaN mode, and NaN handling

I_{FGPN} Software can enable Default NaN mode by setting FPSCR.DN to 1.

I_{DJVH} Using Default NaN mode is a deviation from IEEE 754.

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.

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.

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.

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.

See also:

- *The Default NaN* on page B4-127.
- *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page B4-120.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QMQC	From 8.0	FP	-
NPRL	From 8.0	FP	-
VCSB	From 8.0	FP	-

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

See also:

- *Floating-point encoding formats, half-precision, single-precision, and double-precision* on page B4-120.
- *The Default NaN mode, and NaN handling* on page B4-126.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FQFG	From 8.0	FP	-

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.

R_{LFVH} An Input Denormal exception can occur with other floating-point exceptions.

See also:

- *The IEEE 754 floating-point exceptions* on page B4-123.
- *The Flush-to-zero mode* on page B4-124.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LFVH	From 8.0	FP	-

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.

See also:

- *The IEEE 754 floating-point exceptions* on page B4-123.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PLHJ	From 8.0	FP	-

Chapter B5
Memory Model

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

- *Cache identification* on page B5-174.
- *Cache visibility* on page B5-175.
- *Cache coherency* on page B5-176.
- *Cache enabling and disabling* on page B5-177.
- *Cache behavior at reset* on page B5-178.
- *Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches* on page B5-179.
- *Branch predictors* on page B5-180.
- *Cache maintenance operations* on page B5-181.
- *Ordering of cache maintenance operations* on page B5-184.
- *Branch predictor maintenance operations* on page B5-185.

B5.1 Memory accesses

I_{XRDS}

The memory accesses that are referred to in describing the memory model are instruction fetches from memory and load or store data accesses.

R_{LKQN}

The instruction operation uses the `MemA[]` or `MemU[]` helper functions. If the Main Extension is not implemented unaligned accesses using the `MemU[]` helper functions generate an alignment fault.

R_{BFNF}

A memory access is governed by:

- Whether the access is a read or a write.
- The address alignment.
- Data endianness.
- Memory attributes.

See also:

- *Ordering of implicit memory accesses* on page B5-148.
- *Ordering of explicit memory accesses* on page B5-149.
- *Normal memory* on page B5-154.
- *Device memory* on page B5-157.
- *Memory access restrictions* on page B5-166.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LKQN	From 8.0	None	-
BFNF	From 8.0	None	-

B5.2 Address space

R_{FFMK} The address space is a single, flat address space of 2^{32} bytes.

R_{SNPV} In the address space, byte addresses are unsigned numbers in the range $0\text{-}2^{32}\text{-}1$.

R_{RGBT} If an address calculation overflows or underflows the address space, it wraps around. Address calculations are modulo 2^{32}.

I_{JTKM} Normal sequential execution cannot overflow the top of the address space, because the top of memory always has the Execute Never (XN) memory attribute.

R_{BPMP} One or more accesses that target the top or bottom bytes of memory, or accesses that wrap around the top or bottom, access a sequence of words at increasing memory addresses, effectively incrementing the address by 4 for each load or store. If this calculation overflows the top of the address space, the result is UNPREDICTABLE.

R_{ZXDN} Where an exception entry or tail chaining accesses bytes on the stack that span the top or bottom of the 32-bit memory address space, it is IMPLEMENTATION DEFINED whether stack limit checking is applied.

See also:

- *System address map* on page B6-188.

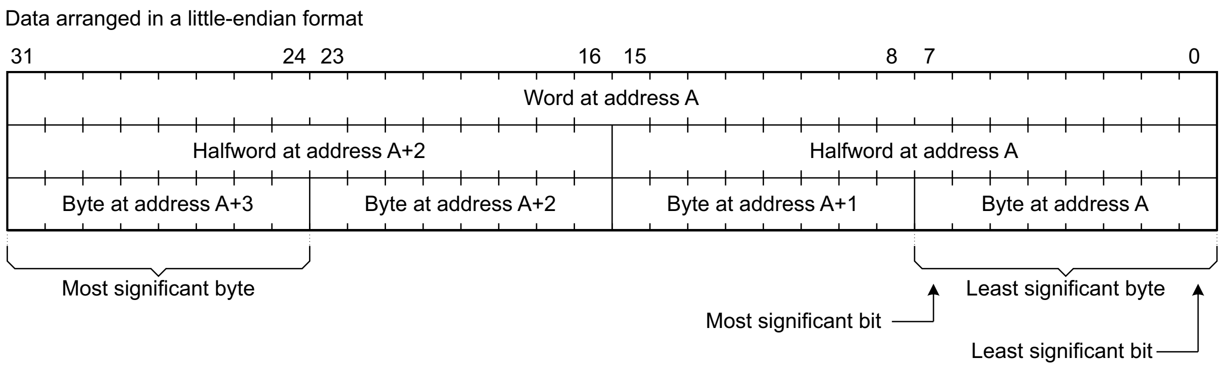
The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

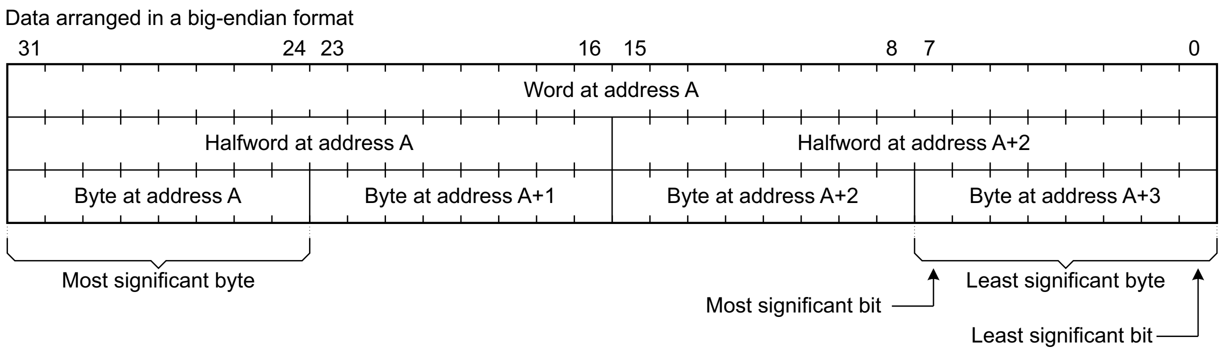
Rule	Architecture version	Required extensions	Notes
FFMK	From 8.0	None	-
SNPV	From 8.0	None	-
RGBT	From 8.0	None	-
BPMP	From 8.0	None	• The encodings of some instructions require M • The encodings of some instructions require FP
ZXDN	From 8.0	None	-

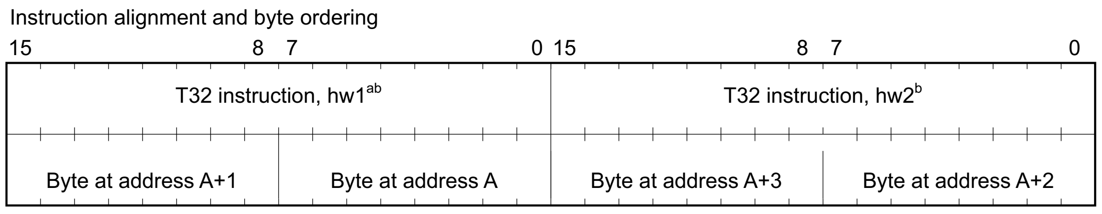
B5.3 Endianness

I_{CTVV} In memory:

- The following figures show the relationship between:
  - The word at address A.
  - The halfwords at addresses A and A+2.
  - The bytes at addresses A, A+1, A+2, and A+3.







a) Bits[15:0]: this is hw 1 for a T32 instruction with a 16-bit encoding

b) Bits[31:0]: this is hw1 and hw2 for a T32 instruction with a 32-bit encoding

R_{JJQL} Instruction fetches are always little-endian, which means that the PE assumes a little-endian arrangement of instructions in memory.

R_{MNSB} All accesses to the Private Peripheral Bus (PPB) are always little-endian, which means that the PE assumes a little-endian arrangement of the PPB registers.

R_{TFKG} The endianness of data accesses is IMPLEMENTATION DEFINED, as indicated by AIRCR.ENDIANNESS.

R_{KPCF} AIRCR.ENDIANNESS is either:

- Implemented with a static value.

- Configured by a hardware input on reset.

R_{XDJV} Instructions that cause a memory access that crosses the PPB boundary are CONSTRAINED UNPREDICTABLE if AIRCR.ENDIANNESS is set to 1. The permitted behavior is one of the following:

- The instruction behaves as a NOP.
- The instruction raises an UNALIGNED UsageFault.
- If the instruction that crossed the PPB boundary was a load, the value of the destination register becomes UNKNOWN.
- If the instruction that crossed the PPB boundary was a store, the value of the memory locations accessed becomes UNKNOWN.

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

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JJQL	From 8.0	None	-
MNSB	From 8.0	None	-
TFKG	From 8.0	None	-
KPCF	From 8.0	None	-
XDJV	From 8.0	None	A UsageFault requires M
QHWC	From 8.0	None	-
XNVS	From 8.0	None	-

B5.4 Alignment behavior

R_{LKGV} All instruction fetches are halfword-aligned.

R_{RQGG} The following are unaligned data accesses that always generate an alignment fault:
- Non halfword-aligned LDAH, LDREXH, LDAEXH, STLH, STLEXH, and STREXH.
- Non word-aligned LDREX, LDAEX, STLEX, STREX, LDRD, LDMIA, LDMDB, POP (multiple registers), LDC, VLDR, VLDM, VPOP, LDA, STL, STMIA, STMDB, PUSH (multiple registers), STC, VSTR, VSTM, VPUSH, VLLDM, and VLSTM.

R_{MHCM} If CCR.UNALIGN_TRP is set to 1, the following are unaligned data accesses that generate an alignment fault:
- Non halfword-aligned LDR{S}H{T}, and STRH{T}.
- Non halfword-aligned TBH.
- Non word-aligned LDR{T}, and STR{T}.

R_{JLGS} Unaligned accesses are only supported if the Main Extension is implemented.

R_{PZTT} If the Main Extension is not implemented, unaligned accesses generate an alignment fault.

R_{WCVX} Accesses to Device memory are always aligned.

R_{RNDS} Alignment faults are synchronous and generate an UNALIGNED UsageFault.

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.

R_{LPVP} Unaligned loads and stores perform the specified load and store to the unaligned memory location.

See also:
- *Normal memory* on page B5-154.
- *Device memory* on page B5-157.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LKGV	From 8.0	None	-
RQGG	From 8.0	None	-
MHCM	From 8.0	None	-
JLGS	From 8.0	M	-
PZTT	From 8.0	!M	-
WCVX	From 8.0	None	-
RNDS	From 8.0	M	-
BNBX	From 8.0	M	-
LPVP	From 8.0	None	-

B5.5 Atomicity

B5.5.1 Single-copy atomicity

I_{NWVK}

Store operations are *single-copy atomic* if, when they overlap bytes in memory:

1. All of the writes from one of the stores are inserted into the coherence order of each overlapping byte.
2. All of the writes from another of the stores are inserted into the coherence order of each overlapping byte.
3. Step 2 repeats, for each single-copy store atomic operation that overlaps.

R$_{\text{BSHJ}}$ The following data accesses are single-copy atomic:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.

R$_{\text{QNPX}}$ Instruction fetches are single-copy atomic at halfword granularity.

R$_{\text{MXWC}}$ For instructions that access a sequence of word-aligned words, each word access is single-copy atomic.

R$_{\text{LKPM}}$ 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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BSHJ	From 8.0	None	-
QNPX	From 8.0	None	-
MXWC	From 8.0	None	-
LKPM	From 8.0	None	-

B5.5.2 Multi-copy atomicity

I$_{\text{BCHK}}$ In a multiprocessing environment, writes to memory are *multi-copy atomic* if all of the following are true:

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

R$_{\text{GJGP}}$ Writes to Normal memory are not required to be multi-copy atomic.

R$_{\text{LBGB}}$ Writes to Device memory with the Gathering attribute are not required to be multi-copy atomic.

R$_{\text{WHJR}}$ Writes to Device memory with the non-Gathering attribute that is single-copy atomic are also multi-copy atomic.

See also:

- *Device memory* on page B5-157.
- *Normal memory* on page B5-154.
- *Load-Acquire and Store-Release accesses to memory* on page B5-170.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
GJGP	From 8.0	None	-
LBGB	From 8.0	None	-
WHJR	From 8.0	None	-

B5.6 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.

R$_{XWVK}$ Unless otherwise stated, concurrent modification and execution of instructions results in a CONSTRAINED UNPREDICTABLE choice of any behavior that can be achieved by executing any sequence of instructions from the same Security state or the same Privilege level.

R$_{BFPB}$ For instructions that can be concurrently modified, the PE executes either:

- The original instruction.
- The modified instruction.

R$_{NNQK}$ A 16-bit instruction can be concurrently modified, where the 16-bit instruction before modification and the 16-bit modification is any of the following:

- `B`.
- `BX`.
- `BLX`.
- `BKPT`.
- `NOP`.
- `SVC`.

R$_{KMZG}$ The hw1 of a 32-bit `BL` immediate instruction can be concurrently modified to the most significant halfword of another `BL` immediate instruction.

R$_{HKGP}$ The hw1 of a 32-bit `BL` immediate instruction can be concurrently modified to a 16-bit `B`, `BLX`, `BKPT`, or `SVC` instruction. This modification also works in reverse.

R$_{FGBT}$ The hw2 of a 32-bit `BL` immediate instruction can be concurrently modified to the hw2 of another `BL` instruction with a different immediate.

R$_{NTVD}$ The hw2, of a 32-bit `B` immediate instruction with a condition field can be concurrently modified to the hw2 of another 32-bit `B` immediate instruction with a condition field with a different immediate.

R$_{CMZX}$ The hw2 of a 32-bit `B` immediate instruction without a condition field can be concurrently modified to the hw2 of another 32-bit `B` immediate instruction without a condition field.

See also:

- *Endianness* on page B5-135.
- *B* on page C2-404.
- *BL* on page C2-412.
- *BLX, BLXNS* on page C2-413.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XWVK	From 8.0	None	-
BFPB	From 8.0	None	-
NNQK	From 8.0	None	-
KMZG	From 8.0	None	-

Rule	Architecture version	Required extensions	Notes
HKGP	From 8.0	None	-
FGBT	From 8.0	None	-
NTVD	From 8.0	None	-
CMZX	From 8.0	None	-

B5.7 Access rights

I~JHGH~ An instruction fetch or memory access is subject to the following checks in the following order:

1. Alignment.
2. SAU.
3. MPU.
4. BusFault (IBUSERR).

R~TQJS~ An exception is generated, instead of normal execution of the fetching and decoding process, if one of the following occurs.

Priority	Fault type	Cause
Highest	One of the following Secure faults: • INVEP • INVTRAN	AU violation
↓	The following MemManage fault: • IACCVIOL	MPU violation
↓	The following BusFault: • IBUSERR	System fault
↓	One of the following: • DebugMonitor exception • Halted Debug Entry	FPB hit
↓	The following SecureFault: • INVEP	SG check
↓	The following UsageFault: • INVSTATE	T32 state check
Lowest	One of the following UsageFaults: • UNDEFINSTR • NOCP	Undefined instruction

R~KPNQ~ If a memory access fails its alignment check, the fetch is not presented to the SAU.

R~SDMQ~ If an instruction fetch or memory access fails its AU check, the fetch is not presented to the relevant MPU for comparison.

R~FLLN~ If an instruction fetch or memory access fails its MPU check, it is not issued to the memory system.

See also:

- *Exception numbers and exception priority numbers* on page B3-53.
- Chapter B8 *The Armv8-M Protected Memory System Architecture*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
TQJS	From 8.0	None	• ASecure fault requires S • An MemManage fault requires M && MPU • A Halted Debug Entry fault can only occur if Halting debug is implemented • A DebugMonitor requires DebugMonitor exception • AUsageFault requires M • ABusFault requires M
KPNQ	From 8.0	S	-
SDMQ	From 8.0	MPU	-
FLLN	From 8.0	MPU	-

B5.8 Observability of memory accesses

R_{PNDH} For a PE, the following mechanisms are treated as independent observers:

- The mechanism that performs reads from or writes to memory.
- The mechanism that causes an instruction cache to be filled from memory or that fetches instructions to be executed directly from memory. These accesses are treated as reads.

R_{DVFW} The set of observers that can observe a memory access is not defined by the PE architecture.

I_{VSCK} In the context of observability, *subsequent* means whichever of the following descriptions is appropriate:

- After the point in time where the location is observed by the observer.
- After the point in time where the location is globally observed.

R_{VCCS} A write to a location in memory is *observed* by an observer when:

- A subsequent read of the location by the same observer would return the value that was written by the observed write or written by a write to that location by any observer that is sequenced in the coherence order of the location after the observed write.
- A subsequent write of the location by the same observer would be sequenced in the coherence order of the location after the observed write.

R_{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.

R_{RSPX} For Device-nGnRnE memory, a read or write of a memory-mapped location in a peripheral is observed, and globally observed, only when the read or write:

- Meets the general observability conditions.
- Can begin to affect the state of the memory-mapped peripheral.
- Can trigger all associated side-effects, whether they affect other peripheral devices, PEs, or memory.

R_{DGRR} A read of a location in memory is *observed* by an observer when a subsequent write to the location by the same observer would have no effect on the value that is returned by the read.

R_{BVJF} A read of a location in memory is *globally observed* for a Shareability domain when a subsequent write to the location by any observer in that Shareability domain that is capable of observing the write would have no effect on the value that is returned by the read.

See also:

- *Device memory* on page B5-157.
- *Device memory attributes* on page B5-159.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PNDH	From 8.0	None	-
DVFW	From 8.0	None	-
VCCS	From 8.0	None	-
XQPT	From 8.0	None	-
RSPX	From 8.0	None	-
DGRR	From 8.0	None	-
BVJF	From 8.0	None	-

B5.9 Completion of memory accesses

R_XCTL A read or write is complete for a Shareability domain when the following conditions are true:

- The read or write is globally observed for that Shareability domain.
- All instruction fetches by observers within the Shareability domain have observed the read or write.

R_WCMQ A cache or branch predictor maintenance instruction is complete for a Shareability domain when the effects of the instruction are globally observed for that Shareability domain.

R_SFLM The completion of a memory access to Device memory other than Device-nGnRnE does not guarantee the visibility of the side-effects of the access to all observers.

R_MWBK The mechanism that ensures the visibility of the side-effects of the access to all observers is IMPLEMENTATION DEFINED.

See also:

- *Shareability domains* on page B5-163.
- *Device memory* on page B5-157.
- *Device memory attributes* on page B5-159.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XCTL	From 8.0	None	-
WCMQ	From 8.0	None	-
SFLM	From 8.0	None	-
MWBK	From 8.0	None	-

B5.10 Ordering requirements for memory accesses

R_{RBDL} Armv8-M defines access restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses involved.

R_{GJDH} For all accesses to all memory types, the only stores by an observer that can be observed by another observer are those stores that have been architecturally executed.

R_{RXPL} Reads and writes can be observed in any order provided that, if an address dependency exists between two reads or between a read and a write, then those memory accesses are observed in program order by all observers within the common Shareability domain of the memory addresses being accessed.

R_{KWFG} Speculative writes by an observer cannot be observed by another observer.

R_{VMHG} For Device memory with the non-Reordering attribute, memory accesses arrive at a single peripheral in program order.

R_{WGCF} Memory accesses caused by instruction fetches are not required to be observed in program order, unless they are separated by a Context synchronization event.

R_{RJMK} A register data dependency between the value that is returned by a load instruction and the address that is used by a subsequent memory transaction enforces an order between that load instruction and the subsequent memory transaction.

See also:

- *Ordering of implicit memory accesses* on page B5-148.
- *Ordering of explicit memory accesses* on page B5-149.
- *Normal memory* on page B5-154.
- *Device memory* on page B5-157.
- *Shareability domains* on page B5-163.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RBDL	From 8.0	None	-
GJDH	From 8.0	None	-
RXPL	From 8.0	None	-
KWFG	From 8.0	None	-
VMHG	From 8.0	None	-
WGCF	From 8.0	None	-
RJMK	From 8.0	None	-

B5.11 Ordering of implicit memory accesses

R_{KPFC} There are no ordering requirements for implicit accesses to any type of memory.

See also:

- *Memory accesses* on page B5-133.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KPFC	From 8.0	None	-

B5.12 Ordering of explicit memory accesses

R_{BMNM} For all memory types, for accesses from a single observer, the requirements of uniprocessor semantics are maintained.

R_{WTRP} For all types of memory, if there is a control dependency between a direct read and a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.

R_{XGNP} For all types of memory, if the value returned by a direct read computes data that is written by a subsequent direct write, the two accesses are observed in program order by any observer in the common Shareability domain of the two accesses.

R_{MBNW} It is impossible for an observer to observe a write from a store that both:

- Has not been executed.
- Will not be executed.

See also:

- *Memory accesses* on page B5-133.
- *Normal memory* on page B5-154.
- *Device memory* on page B5-157.
- *Device memory attributes* on page B5-159.
- *Shareability domains* on page B5-163.
- *Shareability attributes* on page B5-165.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BMNM	From 8.0	None	-
WTRP	From 8.0	None	-
XGNP	From 8.0	None	-
MBNW	From 8.0	None	-

B5.13 Memory barriers

R_{WRCT} The Arm architecture supports out-of-order completion of instructions.

R_{GKDW} Armv8 supports the following memory barriers:

- *Instruction Synchronization Barrier* (`ISB`).
- *Data Memory Barrier* (`DMB`).
- *Data Synchronization Barrier* (`DSB`).

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WRCT	From 8.0	None	-
GKDW	From 8.0	None	-
LQXF	From 8.0	None	-

B5.13.1 Instruction Synchronization Barrier

R_{STMG} An `ISB` ensures that all instructions that come after the `ISB` instruction in program order are fetched from the cache or memory after the `ISB` instruction has completed.

See also:

- `InstructionSynchronizationBarrier()`.
- *Context Synchronization Event* on page B3-110.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
STMG	From 8.0	None	-

B5.13.2 Data Memory Barrier

R_{MPSG} The required Shareability for a `DMB` is *Full system,* and applies to all observers in the Shareability domain.

R_{GVDL} A `DMB` only affects memory accesses and the operation of data cache and unified cache maintenance instructions, and has no effect on the ordering of any other instructions.

R_{HFTX} A `DMB` that ensures the completion of cache maintenance instructions has an access type of both loads and stores.

RWMRT 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.

See also:

- DataMemoryBarrier().
- *Shareability domains* on page B5-163.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MPSG	From 8.0	None	-
GVDL	From 8.0	None	-
HFTX	From 8.0	None	-
WMRT	From 8.0	None	-

B5.13.3 Data Synchronization Barrier

ICNFG The DSB is a memory barrier that synchronizes the execution stream with memory accesses.

RNKWJ The required Shareability for a DSB is Full system and applies to all observers in the Shareability domain.

RVLBF 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.

R_{KMGH} A `DSB` completes when all of the following conditions apply:

- All explicit memory accesses that are observed by PEe before the `DSB` is executed and are of the required access types, and are from observers in the same required Shareability domain as PEe, are complete for the set of observers in the required Shareability domain.
- If the required access types of the `DSB` is reads and writes, then all cache and branch predictor maintenance instructions that are issued by PEe before the `DSB` are complete for the required Shareability domain.
- All explicit accesses to the System Control Space that result in a context altering operation issued by PEe before the `DSB` are complete.

R_{KMBX} No instruction that appears in program order after the `DSB` instruction can execute until the `DSB` completes.

See also:

- `DataSynchronizationBarrier()`.
- *Shareability domains* on page B5-163.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
NKWJ	From 8.0	None	-
VLBF	From 8.0	None	-
KMGH	From 8.0	None	-
KMBX	From 8.0	None	-

B5.13.4 Synchronization requirements for System Control Space

R_{SJQJ} A `DSB` guarantees that all writes to the System Control Space have been completed.

R_{NPDJ} A `DSB` does not guarantee that the side-effects of writes to the System Control Space are visible.

R_{HMNM} A Context synchronization event guarantees that the side-effects of any completed writes to the System Control Space will be visible.

See also:

- *The System Control Space (SCS)* on page B6-192.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
SJQJ	From 8.0	None	-
NPDJ	From 8.0	None	-
HMNM	From 8.0	None	-

B5.14 Normal memory

I_{NVRF}

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.

R$_{QGCF}$ The PE is permitted to treat regions of memory assigned the memory type Normal memory as idempotent.

R$_{CGJX}$ Normal memory can be marked as Cacheable or Non-cacheable. Normal memory is assigned Cacheability attributes.

R$_{LCPJ}$ Normal Non-cacheable memory is always treated as shareable.

R$_{PKXL}$ Speculative data accesses to Normal memory are permitted.

R$_{WLVR}$ A write to Normal memory completes in finite time.

R$_{WLCV}$ A write to a Non-cacheable Normal memory location reaches the endpoint for that location in the memory system in finite time.

R$_{MJWF}$ A completed write to Normal memory is globally observed for the *Shareability domain* in finite time without the requirement for cache maintenance instructions or memory barriers.

R$_{NHFQ}$ For multi-register Load/Store instructions that access Normal memory, the architecture does not define the order in which the registers are accessed.

R$_{CFHV}$ There is no requirement for the memory system beyond the PE to be able to identify the size of the elements accessed.

See also:

- *Memory accesses* on page B5-133.
- *Shareability domains* on page B5-163.
- *Cacheability attributes* on page B5-156.
- *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page B5-169.
- *MAIR_ATTR, Memory Attribute Indirection Register Attributes* on page D1-1081.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QGCF	From 8.0	None	-
CGJX	From 8.0	None	-
LCPJ	From 8.0	None	-
PKXL	From 8.0	None	-
WLVR	From 8.0	None	-

Rule	Architecture version	Required extensions	Notes
WLCV	From 8.0	None	-
MJWF	From 8.0	None	-
NHFQ	From 8.0	None	-
CFHV	From 8.0	None	-

B5.15 Cacheability attributes

R_{KXJV} The architecture provides Cacheability attributes that are defined independently for each of two conceptual levels of cache:

- The Inner cache.
- The Outer cache.

R_{XRWS} The Cacheability attributes are:

- Non-cacheable.
- Write-Through Cacheable.
- Write-Back Cacheable.

R_{XQXW} It is IMPLEMENTATION DEFINED whether Write-Through Cacheable and Write-Back Cacheable can have the additional attribute Transient or Non-transient.

I_{LDXP} The Transient attribute is a memory hint that indicates that the benefit of caching is for a short period. The architecture does not define what is meant by a *short period*.

R_{CFKN} Cacheability attributes other than Non-cacheable can be complemented by the following cache allocation hints, which are independent for read and write accesses:

- Read-Allocate, Transient Read-Allocate, or No Read-Allocate.
- Write-Allocate, Transient Write-Allocate, or No Write-Allocate.

R_{DRTR} The architecture does not require an implementation to make any use of cache allocation hints.

R_{FQSS} Any cacheable Normal memory region is treated as Read-Allocate, No Write-Allocate unless it is explicitly assigned other cache allocation hints.

I_{FRVF} A Cacheable location with no Read-Allocate and no Write-Allocate hints is not the same as a Non-cacheable location. A Non-cacheable location has coherency guarantees for all observers within the system that do not apply to a location that is Cacheable, no Read-Allocate, no Write-Allocate.

R_{FTKW} All data accesses to Non-cacheable Normal memory locations are data coherent to all observers,

See also:

- *Normal memory* on page B5-154.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KXJV	From 8.0	None	-
XRWS	From 8.0	None	-
XQXW	From 8.0	None	-
CFKN	From 8.0	None	-
DRTR	From 8.0	None	-
FQSS	From 8.0	None	-
FTKW	From 8.0	None	-

B5.16 Device memory

I_{BXHS} Device memory is a *memory type* that is assigned to regions of memory where accesses can have side-effects.

R_{WTZL} Device memory is not cacheable.

R_{LDDN} Device memory is always treated as shareable.

R_{PQXS} Speculative data accesses cannot be made to Device memory. However, for instructions that access a sequence of word-aligned words, the accesses might occur multiple times.

R_{NLHC} Speculative instruction fetches can be made to Device memory, unless the location is marked as execute-never.

R_{CSKG} Any unaligned access to Device memory generates an UNALIGNED UsageFault exception.

R_{YMTK} Device memory is assigned a combination of *Device memory attributes*.

R_{LFTG} A write to Device memory completes in finite time.

R_{FSCD} A write to a Device memory location reaches the endpoint for that location in the memory system in finite time.

R_{GTTQ} A completed write to a Device memory location is globally observed for the Shareability domain in finite time without the requirement for cache maintenance instructions or barriers.

R_{XMCH} If the content of a Device memory location changes without a direct write to the location, the change is observed for the Shareability domain in finite time.

R_{KJHG} For an instruction fetch from Device memory, if a branch causes the Program Counter to point to an area of memory that is not marked as execute-never, the implementation can either:

- Treat the fetch as if it is to a location in Normal Non-cacheable memory.
- Take an IACCVIOLL MemManage fault.

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.

R_{KVHT} For an `LDM`, `STM`, `LDRD`, or `STRD` instruction with a register list that includes the PC, the architecture does not define the order in which the registers are accessed.

R_{SFPK} For an `LDM`, `STM`, `VLDM`, 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.

See also:

- *Memory accesses* on page B5-133.
- *Shareability attributes* on page B5-165.
- *Device memory attributes* on page B5-159.
- *Shareability domains* on page B5-163.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WTZL	From 8.0	None	-
LDDN	From 8.0	None	-
PQXS	From 8.0	None	-
NLHC	From 8.0	None	-
CSKG	From 8.0	M	-
YMTK	From 8.0	None	-
LFTG	From 8.0	None	-
FSCD	From 8.0	None	-
GTTQ	From 8.0	None	-
XMCH	From 8.0	None	-
KJHG	From 8.0	None	A MemManage fault requires M && MPU
DFJX	From 8.0	None	-
KVHT	From 8.0	None	-
SFPK	From 8.0	None	-

B5.17 Device memory attributes

R_{VNSJ} Each Device memory region is assigned a combination of Device memory attributes. The attributes are:

Gathering, G and nG

The *Gathering* and *non-Gathering* attributes.

Reordering, R and nR

The *Reordering* and *non-Reordering* attributes.

Early Write Acknowledgement, E and nE

The *Early Write Acknowledgement* and *no Early Write Acknowledgement* attributes.

R$_{CFFC}$ Each Device memory region is assigned one of the combinations in the following table:

Memory Ordering	Name	nG	nR	nE	G	R	E
Strong	Device-nGnRnE	Y	Y	Y	-	-	-
↓	Device-nGnRE	Y	Y	-	-	-	Y
↓	Device-nGRE	Y	-	-	-	Y	Y
Weak	Device-GRE	-	-	-	Y	Y	Y

R$_{LJKD}$ Weaker memory can be accessed according to the rules specified for stronger memory:

- Memory with the:
  - G attribute can be accessed according to the rules specified for the nG attribute.
  - nG attribute cannot be accessed according to the rules specified for the G attribute.
- Memory with the:
  - R attribute can be accessed according to the rules specified for the nR attribute.
  - nR attribute cannot be accessed according to the rules specified for the R attribute.

Because the nE attribute is a hint:

- An implementation is permitted to perform an access with the E attribute in a manner consistent with the requirements specified by the nE attribute.
- An implementation is permitted to perform an access with the nE attribute in a manner consistent with the relaxations allowed by the E attribute.

R$_{FJXX}$ For Device-GRE and Device-nGRE memory, the use of barriers is required to order accesses.

See also:

- *Gathering and non-Gathering Device memory attributes* on page B5-160.
- *Reordering and non-Reordering Device memory attributes* on page B5-161.
- *Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes* on page B5-162.
- *Device memory* on page B5-157.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VNSJ	From 8.0	None	-
CFFC	From 8.0	None	-
LJKD	From 8.0	None	-
FJXX	From 8.0	None	-

B5.17.1 Gathering and non-Gathering Device memory attributes

G attribute

R_{DBSX} If multiple accesses of the same type, read or write, are to:

- The same location, with the G attribute, they can be merged into a single transaction.
- Different locations, all with the G attribute, they can be merged into a single transaction.

R_{KCMX} Gathering of accesses that are separated by a memory barrier is not permitted.

R_{JSRD} Gathering of accesses that are generated by a Load-Acquire/Store-Release is not permitted.

R_{MGKJ} A read can come from intermediate buffering of a previous write if:

- The accesses are not separated by a `DMB` or `DSB` barrier.
- The accesses are not separated by any other ordering construction that requires that the accesses are in order, for example a combination of Load-Acquire and Store-Release.
- The accesses are not generated by a Store-Release instruction.

I_{SRDS} The architecture only defines programmer visible behavior. Therefore, if a programmer cannot tell whether Gathering has occurred, Gathering can be performed.

nG attribute

R_{GVTF} Multiple accesses to a memory location with the nG attribute cannot be merged into a single transaction.

R_{BTWD} A read of a memory location with the nG attribute cannot come from a cache or a buffer, but comes from the endpoint for that address in the memory system.

See also:

- *Load-Acquire and Store-Release accesses to memory* on page B5-170.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DBSX	From 8.0	None	-
KCMX	From 8.0	None	-
JSRD	From 8.0	None	-
MGKJ	From 8.0	None	-
GVTF	From 8.0	None	-
BTWD	From 8.0	None	-

B5.17.2 Reordering and non-Reordering Device memory attributes

R attribute

R_{RPTB} This attribute imposes no restrictions or relaxations.

nR attribute

R_{DFXL} If the access is to a:

- Peripheral, it arrives at the peripheral in program order. If there is a mixture of accesses to Device nGnRE and Device-nGnRnE in the same peripheral, these accesses occur in program order.
- Non-peripheral, this attribute imposes no restrictions or relaxations.

I_{BDWB} The IMPLEMENTATION DEFINED size of the single peripheral is the same as applies for the ordering guarantee that is provided by the `DMB` instruction.

R_{NDHC} The non-Reordering attribute does not require any additional ordering, other than the ordering that applies to Normal memory, between:

- Accesses with the non-Reordering attribute and accesses with the Reordering attribute.
- Accesses with the non-Reordering attribute and accesses to Normal memory.
- Accesses with the non-Reordering attribute and accesses to different peripherals of IMPLEMENTATION DEFINED size.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RPTB	From 8.0	None	-
DFXL	From 8.0	None	-
NDHC	From 8.0	None	-

B5.17.3 Early Write Acknowledgement and no Early Write Acknowledgement Device memory attributes

E attribute

R_{PVSH} This attribute imposes no restrictions or relaxations.

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.

I_{FQWQ} The E attribute is treated as a hint. Arm strongly recommends that this hint is not ignored by a PE, but is made available for use by the system.

See also:

- *Memory barriers* on page B5-150

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

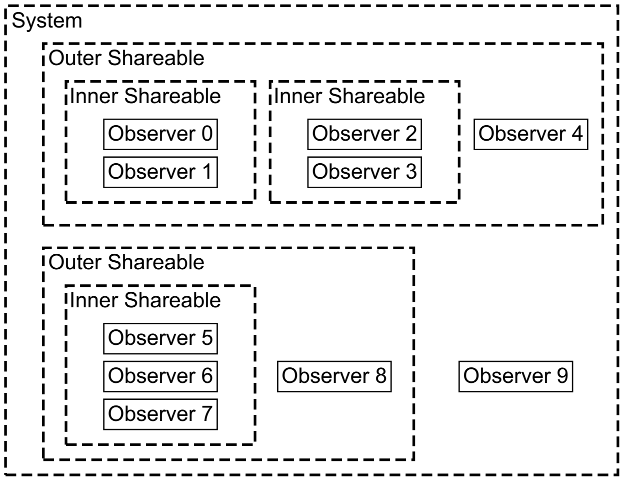
Rule	Architecture version	Required extensions	Notes
PVSH	From 8.0	None	-
FWFR	From 8.0	None	-

B5.18 Shareability domains

R_{JMHL} There are two conceptual Shareability domains:

- The Inner Shareability domain.
- The Outer Shareability domain.

I_{XQWM} The following diagram shows the Shareability domains:



R_{MCPS} All observers in an Inner Shareability domain are data coherent for data accesses to memory that has the *Inner-shareable Shareability attribute*.

R_{SVCR} All observers in an Outer Shareability domain are data coherent for data accesses to memory that has the *Outer-shareable Shareability attribute*.

R_{JMFS} Each observer is a member of only a single Inner Shareability domain.

R_{BNWH} Each observer is a member of only a single Outer Shareability domain.

R_{FVBG} All members of the same Inner Shareability domain are always members of the same Outer Shareability domain.

R_{WFMV} Accesses to a shareable memory location are coherent within the Shareability domain of that location.

I_{DHJF} An Inner Shareability domain is a subset of an Outer Shareability domain, although it is not required to be a proper subset.

R_{XHJL} Hardware is required to ensure coherency and ordering within the Shareability domain if all of the following apply:

- Before writing to a location not using the Write-Back attribute, a location in the caches that might have been written with the Write-Back attribute by an agent has been invalidated or cleaned.
- After writing the location with the Write-Back attribute, the location has been cleaned from the caches to make the write visible to external memory.
- Before reading the location with a cacheable attribute, the cache location has been invalidated, or cleaned and invalidated.
- A `DMB` barrier instruction has been executed, with a scope that applies to the common Shareability of the accesses, between any accesses to the same memory location that use different attributes.

See also:

- *Observability of memory accesses* on page B5-144.
- *Shareability attributes* on page B5-165.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JMHL	From 8.0	None	-
MCPS	From 8.0	None	-
SVCR	From 8.0	None	-
JMFS	From 8.0	None	-
BNWH	From 8.0	None	-
FVBG	From 8.0	None	-
WFMV	From 8.0	None	-
XHJL	From 8.0	None	-

B5.19 Shareability attributes

R$_{\text{CJRF}}$ Each Normal cacheable memory region is assigned one of the following Shareability attributes:

- *Non-shareable*.
- *Inner-shareable*.
- *Outer-shareable*.

R$_{\text{PDVV}}$ 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.

R$_{\text{XTVD}}$ Non-cacheable Normal memory locations are always treated as Outer Shareable.

See also:

- *Memory accesses* on page B5-133.
- *Normal memory* on page B5-154.
- *Device memory* on page B5-157.
- *Shareability domains* on page B5-163.
- *Cache maintenance operations* on page B5-181.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CJRF	From 8.0	None	-
PDVV	From 8.0	None	-
XTVD	From 8.0	None	-

B5.20 Memory access restrictions

R_{KSXT} For accesses to any two bytes that are accessed by the same instruction, the two bytes have the same memory type and Shareability attributes, otherwise behavior is a CONSTRAINED UNPREDICTABLE choice of the following:

- All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the first address that is accessed by the instruction.
- All memory accesses that were generated by the instruction use the memory type and Shareability attributes that are associated with the last address that is accessed by the instruction.
- Each memory access that is generated by the instruction uses the memory type and Shareability attribute that is associated with its own address.
- The instruction executes as a NOP.
- The instruction generates an alignment fault caused by the memory type.

I_{WRBT} Except for possible differences in cache allocation hints, Arm deprecates having different Cacheability attributes for accesses to any two bytes that are generated by the same instruction.

R_{BFKS} If the accesses of an instruction that cause multiple accesses to any type of Device memory cross the boundary of a memory region then the behavior is a CONSTRAINED UNPREDICTABLE choice of the following:

- All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses.
- All memory accesses that are generated by the instruction are performed as if the presence of the boundary had no effect on memory accesses, except that there is no guarantee of ordering between memory accesses,
- The instruction executes as a NOP.
- The instruction generates an alignment fault caused by the memory type.

See also:

- *Memory accesses* on page B5-133.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KSXT	From 8.0	None	-
BFKS	From 8.0	None	-

B5.21 Mismatched memory attributes

R_{XHTK} Memory locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of all the following memory attributes of that location:

- Memory type - Device or Normal.
- Shareability.
- Cacheability, for the same level of the Inner or Outer cache, but excluding any cache allocation hints.

R_{VKHJ} When a memory location is accessed with mismatched attributes, the only permitted effects are one or more of the following:

- Uniprocessor semantics for reads and writes to that memory location might be lost. This means:
  - A read of the memory location by one agent might not return the value that was most recently written to that memory location by the same agent.
  - Multiple writes to the memory location by one agent with different memory attributes might not be ordered in program order.
- There might be a loss of coherency when multiple agents attempt to access a memory location.
- There might be a loss of the properties that are derived from the memory type.
- If all Load-Exclusive/Store-Exclusive instructions that are executed across all threads to access a given memory location do not use consistent memory attributes, the exclusive monitor state becomes UNKNOWN.
- Bytes that are written without the Write-Back cacheable attribute and that are within the same Write-Back granule as bytes that are written with the Write-Back cacheable attribute might have their values reverted to the old values as a result of cache Write-Back.

R_{NJLB} The loss of the properties that are associated with mismatched memory type attributes refers only to the following properties of Device memory that are additional to the properties of Normal memory:

- Prohibition of speculative read accesses.
- Prohibition on Gathering.
- Prohibition on Reordering.

R_{QCKK} If the only memory type mismatch that is associated with a memory location across all users of the memory location is between different types of Device memory, then all accesses might take the properties of the weakest Device memory type.

R_{HCCD} Any agent that reads a memory location with mismatched attributes using the same common definition of the Shareability and Cacheability attributes is guaranteed to access it coherently, to the extent required by that common definition of the memory attributes, only if all the following conditions are met:

- All aliases to the memory location with write permission both use a common definition of the Shareability and Cacheability attributes for the memory location, and have the Inner Cacheability attribute the same as the Outer Cacheability attribute.
- All aliases to a memory location use a definition of the Shareability attributes that encompasses all the agents with permission to access the location.

R_{GBKH} The possible permitted effects that are caused by mismatched attributes for a memory location are defined more precisely if all the mismatched attributes define the memory location as one of:

- Any Device memory type.
- Normal Inner Non-cacheable, Outer Non-cacheable memory.

In these cases, the only permitted software-visible effects of the mismatched attributes are one or more of the following:

- Possible loss of properties that are derived from the memory type when multiple agents attempt to access the memory location.
- Possible reordering of memory transactions to the same memory location with different memory attributes, potentially leading to a loss of coherency or uniprocessor semantics. Any possible loss of coherency or uniprocessor semantics can be avoided by inserting `DMB` barrier instructions between accesses to the same memory location that might use different attributes.

R$_{VVBS}$ If the mismatched attributes for a location mean that multiple cacheable accesses to the location might be made with different Shareability attributes, then ordering and coherency are guaranteed only if:

- Each PE that accesses the location with a cacheable attribute performs a clean and invalidate of the location before and after accessing that location.
- A `DMB` barrier with scope that covers the full Shareability of the accesses is placed between any accesses to the same memory location that use different attributes.

R$_{VCXW}$ If multiple agents attempt to use Load-Exclusive or Store-Exclusive instructions to access a location, and the accesses from the different agents have different memory attributes associated with the location, the exclusive monitor state becomes UNKNOWN.

I$_{TPWG}$ Arm strongly recommends that software does not use mismatched attributes for aliases of the same location. An implementation might not optimize the performance of a system that uses mismatched aliases.

See also:
- Chapter B8 *The Armv8-M Protected Memory System Architecture*.
- *Shareability domains* on page B5-163.
- *Cacheability attributes* on page B5-156.
- *Device memory* on page B5-157.
- *Normal memory* on page B5-154.
- *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page B5-169.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XHTK	From 8.0	None	-
VKHJ	From 8.0	None	-
NJLB	From 8.0	None	-
QCKK	From 8.0	None	-
HCCD	From 8.0	None	-
GBKH	From 8.0	None	-
VVBS	From 8.0	None	-
VCXW	From 8.0	None	-

B5.22 Load-Exclusive and Store-Exclusive accesses to Normal memory

R_{KDWC} For Normal memory that is:

- Non-shareable, it is IMPLEMENTATION DEFINED whether Load-Exclusive and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.
- Shareable, Load-Exclusive, and Store-Exclusive instructions take account of the possibility of accesses by more than one observer.

See also:

- *Normal memory* on page B5-154.
- *Memory accesses* on page B5-133.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KDWC	From 8.0	None	-

B5.23 Load-Acquire and Store-Release accesses to memory

I_{VVTX}

The following table summarizes the Load-Acquire/Store-Release instructions.

Data type	Load-Acquire	Store-Release	Load-Acquire Exclusive	Store-Release Exclusive
32-bit word	`LDA`	`STL`	`LDAEX`	`STLEX`
16-bit halfword	`LDAH`	`STLH`	`LDAEXH`	`STLEXH`
8-bit byte	`LDAB`	`STLB`	`LDAEXB`	`STLEXB`

R_{XBRM} A Store-Release followed by a Load-Acquire is observed in program order by each observer within the Shareability domain of the memory address being accessed by the Store-Release and the memory address being accessed by the Load-Acquire.

R_{RRFK} For a Load-Acquire, observers in the Shareability domain of the address that is accessed by the Load-Acquire observe accesses in the following order:

1. The read caused by the Load-Acquire.
2. Reads and writes caused by loads and stores that appear in program order after the Load-Acquire for which the Shareability of the address that is accessed by the load or store requires that the observer observes the access.

There are no other ordering requirements on loads or stores that appear before the Load-Acquire.

R_{WLWT} For a Store-Release, observers in the Shareability domain of the address that is accessed by the Store-Release observe accesses in the following order:

1. All of the following for which the Shareability of the address that is accessed requires that the observer observes the access:
   - Reads and writes caused by loads and stores that appear in program order before the Store-Release.
   - Writes that were observed by the PE executing the Store-Release before it executed the Store-Release.
2. The write caused by the Store-Release.

There are no other ordering requirements on loads or stores that appear in program order after the Store-Release.

R_{HCKC} All Store-Release instructions are multi-copy atomic when they are observed with Load-Acquire instructions.

R_{DGXR} A Load-Acquire to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed after the Load-Acquire will arrive at the memory-mapped peripheral after the memory access of the Load-Acquire.

R_{CKRC} A Store-Release to an address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access ensures that all memory accesses using Device memory types to the same memory-mapped peripheral that are architecturally required to be observed before the Store-Release will arrive at the memory-mapped peripheral before the memory access of the Store-Release.

R_{GJHK} If a Load-Acquire to a memory address in a memory-mapped peripheral of an arbitrary system-defined size that is defined as any type of Device memory access has observed the value that is stored to that address by a Store-Release, then any memory access to the memory-mapped peripheral that is architecturally required to be ordered before the memory access of the Store-Release will arrive at the memory-mapped peripheral before any memory access to the same peripheral that is architecturally required to be ordered after the memory access of the Load-Acquire.

R_{WRLC} Load-Acquire and Store-Release access only a single data element.

R_{KCTN} Load-Acquire and Store-Release accesses are single-copy atomic.

R_{BXRP} If a Load-Acquire or Store-Release instruction accesses an address that is not aligned to the size of the data element being accessed, the access generates an alignment fault.

R$_{NVRJ}$ A Store-Release Exclusive instruction only has the release semantics if the store is successful.

See also:
- *Shareability domains* on page B5-163.
- *Device memory* on page B5-157.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XBRM	From 8.0	None	-
RRFK	From 8.0	None	-
WLWT	From 8.0	None	-
HCKC	From 8.0	None	-
DGXR	From 8.0	None	-
CKRC	From 8.0	None	-
GJHK	From 8.0	None	-
WRLC	From 8.0	None	-
KCTN	From 8.0	None	-
BXRP	From 8.0	None	-
NVRJ	From 8.0	None	-

B5.24 Caches

I_{JSPB} When a memory location is marked with a Normal Cacheable memory attribute, determining whether a copy of the memory location is held in a cache can depend on many aspects of the implementation, such as the following factors:

- The size, line length, and associativity of the cache.
- The cache allocation algorithm.
- Activity by other elements of the system that can access the memory.
- Speculative instruction fetching algorithms.
- Speculative data fetching algorithms.
- Interrupt behaviors.

R_{QGSQ} An implementation can include multiple levels of cache, up to a maximum of seven levels, in a hierarchical memory system.

I_{STRV} The lower the cache level, the closer the cache is to the PE.

R_{PDSR} Entries for addresses with a Normal cacheable attribute can be allocated to an enabled cache at any time.

R_{JGBL} The allocation of a memory address to a cache location is IMPLEMENTATION DEFINED.

R_{SBGJ} A cache entry covers at least 16 bytes and no more than 2KB of contiguous address space, aligned to its size.

R_{XXBW} Where a breakdown in coherency can occur, data coherency of the caches is controlled in an IMPLEMENTATION DEFINED manner.

R_{JVJN} The architecture cannot guarantee whether:

- A memory location that is present in the cache remains in the cache.
- A memory location that is not present in the cache is brought into the cache.

R_{PHWM} If the cache is disabled, no new allocation of memory locations into the cache occurs.

R_{LJQB} The allocation of a memory location into a cache cannot cause the most recent value of that memory location to become invisible to an observer, if it had previously been visible to that observer.

R_{QRLS} If the cache is enabled, it is guaranteed that no memory location that does not have a cacheable attribute is allocated into the cache.

R_{XXVH} If the cache is enabled, it is guaranteed that no memory location is allocated to the cache if the access permissions for that location are so that the location cannot be accessed by reads and cannot be accessed by writes.

R_{SCKQ} Any cached memory location is not guaranteed to remain incoherent with the rest of memory.

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.

R_{WDBP} It is UNPREDICTABLE whether the location is returned from cache or from memory when:

- The location is not marked as cacheable but is contained in the cache. This situation can occur if a location is marked as Non-cacheable after it has been allocated into the cache.
- The location is marked as cacheable and might be contained in the cache, but the cache is disabled.

R_{NDNN} The architecture allows copies of control values or data values to be cached. The existence of such copies can lead to CONSTRAINED UNPREDICTABLE behavior, if the cache has not been correctly invalidated following a change of the control or data values.

Unless explicitly stated otherwise, the behavior of the PE is consistent with:

- The old value.
- The new value.
- An amalgamation of the old and new values.

I_{BMPQ} The choice between the behaviors might, in some implementations, vary for each use of a control or data value.

See also:

- *Cache identification* on page B5-174.
- *Cache enabling and disabling* on page B5-177.
- *Cacheability attributes* on page B5-156.
- *Cache behavior at reset* on page B5-178.
- *Ordering of cache maintenance operations* on page B5-184.
- *Mismatched memory attributes* on page B5-167.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QGSQ	From 8.0	None	-
PDSR	From 8.0	None	-
JGBL	From 8.0	None	-
SBGJ	From 8.0	None	-
XXBW	From 8.0	None	-
JVJN	From 8.0	None	-
PHWM	From 8.0	None	-
LJQB	From 8.0	None	-
QRLS	From 8.0	None	-
XXVH	From 8.0	None	-
SCKQ	From 8.0	None	-
RQXN	From 8.0	None	-
WDBP	From 8.0	None	-
NDNN	From 8.0	None	-

B5.25 Cache identification

R_{WBGH} A PE controls the implemented caches using:

- A single Cache Type Register, CTR.
- A single Cache Level ID Register, CLIDR.
- A single Cache Size Selection Register, CSSELR.
- For each implemented cache, across all levels of caching, a Cache Size Identification Register, CCSIDR.

R_{XJTL} The number of levels of cache is IMPLEMENTATION DEFINED and can be determined from the Cache Level ID Register.

I_{PPSB} Cache sets and Cache ways are numbered from 0. Usually the set number is an IMPLEMENTATION DEFINED function of an address.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WBGH	From 8.0	None	-
XJTL	From 8.0	None	-

B5.26 Cache visibility

R~QLVB~ A completed write to a memory location that is Non-cacheable or Write-Through Cacheable for a level of cache made by an observer accessing the memory system inside the level of cache is visible to all observers accessing the memory system outside the level of cache without the need of explicit cache maintenance.

R~RCHC~ A completed write to a memory location that is Non-cacheable for a level of cache made by an observer accessing the memory system outside the level of cache is visible to all observers accessing the memory system inside the level of cache without the need of explicit cache maintenance.

See also:

- *Cacheability attributes* on page B5-156.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QLVB	From 8.0	None	-
RCHC	From 8.0	None	-

B5.27 Cache coherency

R_{NNDJ} Data coherency of caches is ensured:

- When caches are not used.
- As a result of cache maintenance operations.
- By the use of hardware coherency mechanisms to ensure coherency of data accesses to memory for cacheable locations by observers in different Shareability domains.

R_{CPGW} Hardware is not required to ensure coherency between instruction caches and memory, even for regions of memory with the Shareability attribute.

See also:

- *Cache maintenance operations* on page B5-181.
- *Memory barriers* on page B5-150.
- *Shareability attributes* on page B5-165.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
NNDJ	From 8.0	None	-
CPGW	From 8.0	None	-

B5.28 Cache enabling and disabling

IPPLL The Configuration and Control Register, CCR, enables and disables caches across all levels of cache that are visible to the PE.

RHTLD It is IMPLEMENTATION DEFINED whether the CCR.DC and CCR.IC bits affect the memory attributes that are generated by an enabled MPU.

ITNHX An implementation can use control bits in the Auxiliary Control Register, ACTLR, for finer-grained control of cache enabling.

RDSTQ 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.

See also:
- *Cache identification* on page B5-174.
- *Caches* on page B5-172.
- *Cache behavior at reset* on page B5-178.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HTLD	From 8.0	M && MPU	-
DSTQ	From 8.0	M && MPU	-

B5.29 Cache behavior at reset

R_{KCFK} All caches are disabled at reset.

R_{JMBT} An implementation can require the use of a specific cache initialization routine to invalidate its storage array before it is enabled:

- The exact form of any required cache initialization routine is IMPLEMENTATION DEFINED.
- If a required initialization routine is not performed, the state of an enabled cache is UNPREDICTABLE.

R_{TVKQ} If an implementation permits cache hits when the cache is disabled, the cache initialization routine provides a mechanism to ensure the correct initialization of the caches.

R_{CJGV} If an implementation permits cache hits when the cache is disabled and the cache contents are not invalidated at reset, the initialization routine avoids any possibility of running from an uninitialized cache.

I_{JSQQ} An initialization routine can require a fixed instruction sequence to be placed in a restricted range of memory.

I_{JCTD} Arm recommends that whenever an invalidation routine is required, it is based on the Armv8-M cache maintenance operations.

See also:
- *Caches* on page B5-172.
- *Cache enabling and disabling* on page B5-177.
- *Cache maintenance operations* on page B5-181.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KCFK	From 8.0	None	-
JMBT	From 8.0	None	-
TVKQ	From 8.0	None	-
CJGV	From 8.0	None	-

B5.30 Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches

I_{CQLR} PLD and PLI are memory system hints and their effect is IMPLEMENTATION DEFINED.

I_{TPPK} The instructions do not generate exceptions but the memory system operations might generate an imprecise fault (asynchronous exception) because of the memory access.

R_{QNGJ} A PLD instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the equivalent load from the same location with the same context and at the same privilege level and Security state.

R_{SFNK} A PLD instruction does not access Device-nGnRnE or Device-nGnRE memory.

R_{HNLN} A PLI instruction does not cause any effect to the caches or memory other than the effects that, for permission or other reasons, can be caused by the fetch resulting from changing the PC to the location specified by the PLI instruction with the same context and at the same privilege level and Security state.

R_{MRFG} A PLI instruction cannot access memory that has the Device-nGnRnE or Device-nGnRE attribute.

See also:
- PLD, PLDW (immediate).
- PLD (literal).
- PLD (register).
- PLI (immediate, literal).
- PLI (register).

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QNGJ	From 8.0	None	-
SFNK	From 8.0	None	-
HNLN	From 8.0	None	-
MRFG	From 8.0	None	-

B5.31 Branch predictors

I_{GTPB} Branch predictor hardware typically uses a form of cache to hold branch information.

R_{MTBD} Branch predictors are not architecturally visible.

I_{CVCV} The BPIALL operation is provided for timing and determinism

See also:

- *Branch predictor maintenance operations* on page B5-185.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MTBD	From 8.0	None	-

B5.32 Cache maintenance operations

I_{MRMG} Cache maintenance operations act on particular memory locations.

R_{JJLL} Following a Clean operation, updates made by an observer that controls the cache are made visible to other observers that can access memory at the point to which the operation is performed.

R_{VRBP} The cleaning of a cache entry from a cache can overwrite memory that has been written by another observer only if the entry contains a location that has been written to by an observer in the Shareability domain of that memory location.

R_{SJFS} Following an invalidate operation, updates made visible by observers that access memory at the point to which the invalidate is defined are made visible to an observer that controls the cache.

R_{PGXK} An invalidate operation might result in the loss of updates to the locations affected by the operation that have been written by observers that access the cache.

R_{TKBD} If the address of an entry on which the invalidate operates does not have a Normal cacheable attribute, or if the cache is disabled, then an invalidate operation ensures that this address is not present in the cache.

R_{JTXK} If the address of an entry on which the invalidate operates has the Normal cacheable attribute, the cache invalidate operation cannot ensure that the address is not present in an enabled cache.

R_{SDVP} A clean and invalidate operation behaves as the execution of a clean operation followed immediately by an invalidate operation. Both operations are performed to the same location.

R_{VKSN} The clean operation cleans from the level of cache that is specified through at least the next level of cache away from the PE.

R_{GFXB} The invalidate operation invalidates only at the level specified.

R_{KVSM} For set/way operations and for All (entire cache) operations, the cache maintenance operation is to the next level of caching.

R_{JTWT} For address operations, the cache maintenance operation is to the point of coherency (PoC) or to the point of unification (PoU) depending on the settings in CLIDR.LoC and CLIDR.LOUU.

R_{XLHX} Data cache maintenance operations affect data caches and unified caches.

R_{QKMF} Instruction cache maintenance operations only affect instruction caches.

R_{RSVL} Cache maintenance operations are memory mapped, 32-bit write-only operations.

R_{NSHH} Cache maintenance operations can have one of the following side-effects:

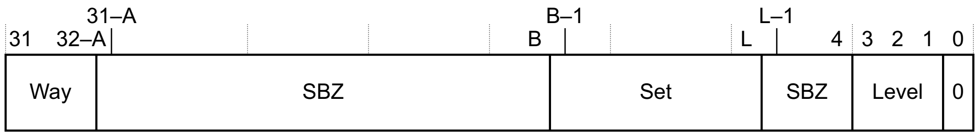
- Any location in the cache might be cleaned.
- Any unlocked location in the cache might be cleaned and invalidated.

R_{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.

R_{HCTC} For `DCISW`, `DCCSW`, and `DCCISW`, the `STR` operation identifies the cache line to which it applies by specifying the following:

- The cache set the line belongs to.
- The way number of the line in the set.
- The cache level.

The format of the register data for a set/way operation is:



Where:

A	= Log2(ASSOCIATIVITY), rounded up to the next integer if necessary.
B	= (L + S).
L	= Log2(LINELEN).
S	= Log2(NSETS), rounded up to the next integer if necessary. ASSOCIATIVITY, LINELEN (line length, in bytes), and NSETS (number of sets) have their usual meanings and are the values for the cache level being operated on. The values of A and S are rounded up to the next integer.
Level	((Cache level to operate on)–1). For example, this field is 0 for operations on an L1 cache, or 1 for operations on an L2 cache.
Set	The number of the set to operate on.
Way	The number of the way to operate on.

- If L == 4 then there is no SBZ field between the set and level fields in the register.
- If A == 0 there is no way field in the register, and register bits[31:B] are SBZ.
- If the level, set, or way field in the register is larger than the size implemented in the cache, then the effect of the operation is UNPREDICTABLE.

R_{RSBX} After the completion of an instruction cache maintenance operation, a Context synchronization event guarantees that the effects of the cache maintenance operation are visible to all instruction fetches that follow the Context synchronization event.

I_{DHJQ} Arm recommends that, wherever possible, all caches that require maintenance to ensure coherency are included in the caches affected by the architecturally-defined cache maintenance operations.

R_{LRGS} It is IMPLEMENTATION DEFINED whether the `DCIMVAC` and `DCISW` operations, when performed from Non-secure state either:

- Clean any data that might be Secure data before invalidating it.
- Do not invalidate Secure data.

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.

I_{MLLC} The following is the sequence of cache cleaning operations for a line of self-modifying code.

```
; Enter this code with <Rx> containing the new 32-bit instruction and <Ry>;
containing the address of the instruction.
; Use STRH in the first line instead of STR for a 16-bit instruction.
STR <Rx>, [<Ry>] ; Write instruction to memory
DSB ; Ensure write is visible
MOV <Rt>, 0xE000E000 ; Create pointer to base of System Control Space
STR <Ry>, [<Rt>,#0xF64] ; Clean data cache by address to point of unification
DSB ; Ensure visibility of the data cleaned from the cache
STR <Ry>, [<Rt>,#0xF58] ; Invalidate instruction cache by address to PoU
STR <Ry>, [<Rt>,#0xF78] ; Invalidate branch predictor
DSB ; Ensure completion of the invalidations
ISB ; Synchronize fetched instruction stream
```

R_{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.

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.

See also:
- *Cache Maintenance Operations* on page D1-867.
- *Cache Maintenance Operations (NS alias)* on page D1-872.
- *Observability of memory accesses* on page B5-144.
- *Cacheability attributes* on page B5-156.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JJLL	From 8.0	None	-
VRBP	From 8.0	None	-
SJFS	From 8.0	None	-
PGXK	From 8.0	None	-
TKBD	From 8.0	None	-
JTXK	From 8.0	None	-
SDVP	From 8.0	None	-
VKSN	From 8.0	None	-
GFXB	From 8.0	None	-
KVSM	From 8.0	None	-
JTWT	From 8.0	None	-
XLHX	From 8.0	None	-
RSVL	From 8.0	None	-
NSHH	From 8.0	None	-
DWMR	From 8.0	None	-
HCTC	From 8.0	None	-
RSBX	From 8.0	None	-
LRGS	From 8.0	S	-
VKDF	From 8.0	S	-
HXMM	From 8.0	S	-
JFGF	From 8.0	None	A BusFault requires M

B5.33 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.

RGXNL 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.

RRTJG There is no restriction on the ordering of data or unified cache maintenance operation by address relative to any explicit load or store.

RMJPP 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.

IVXXZ A `DSB` instruction can be inserted to enforce ordering as required.

RSWBG For the `ICIALLU` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.

IZQQZ In a PE with the Security Extension, if cache maintenance operations are required when the security attribution of memory is changed, the following sequence of steps can be followed:

1. If the attribution of the address range changes from Secure to Non-secure, ensure that memory does not contain any data that is to remain secure.
2. Execute a `DSB` instruction.
3. Clean the affected lines in data or unified caches using the `DCC*` instruction.
4. Execute a `DSB` instruction.
5. Change the security attribution of the address range.
6. Execute a `DSB` instruction.
7. Invalidate the affected lines in all caches using the `DCI*` and `ICI*` instructions.
8. Execute a Context synchronization event.

See also:

- *Data Synchronization Barrier* on page B5-151.
- *Security attribution* on page B8-209.
- *Cache maintenance operations* on page B5-181.
- *Security attribution* on page B8-209.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GCNB	From 8.0	None	-
GXNL	From 8.0	None	-
RTJG	From 8.0	None	-
MJPP	From 8.0	None	-
SWBG	From 8.0	None	-

B5.34 Branch predictor maintenance operations

R_{HVXX} Branch predictor maintenance operations are independent of cache maintenance operations.

R_{NSRK} A Context synchronization event that follows a branch predictor maintenance operation guarantees that the effects of the branch predictor maintenance operation are visible to all instructions after the context synchronization event.

R_{HRXF} For the `BPIALL` operation, the value in the register specified by the `STR` instruction that performs the operation is ignored.

R_{LXHX} As a side-effect of a branch predictor maintenance operation, any entry in the branch predictor might be invalidated.

See also:
- *Cache Maintenance Operations* on page D1-867.
- *Cache Maintenance Operations (NS alias)* on page D1-872.
- *BPIALL, Branch Predictor Invalidate All* on page D1-887.
- *Memory barriers* on page B5-150.
- *DSB* on page C2-432.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HVXX	From 8.0	None	-
NSRK	From 8.0	None	-
HRXF	From 8.0	None	-
LXHX	From 8.0	None	-

Chapter B6
The System Address Map

This chapter specifies the Armv8-M system address map rules. It contains the following sections:

B6.1 System address map

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[a]RA[b]	Non-shareable	Typically ROM or flash memory.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	WBWA[c]RA	Non-shareable	SRAM region typically used for on-chip RAM.
0x40000000-0x5FFFFFFF	Peripheral	Device, nGnRE	XN[d]	-	Shareable	On-chip peripheral address space.
0x60000000-0x7FFFFFFF	RAM	Normal	-	WBWARA	Non-shareable	Memory with write-back, write allocate cache attribute for L2/L3 cache support.
0x80000000-0x9FFFFFFF	RAM	Normal	-	WTRA	Non-shareable	Memory with Write-Through cache attribute.
0xA0000000-0xBFFFFFFF	Device	Device, nGnRE	XN	-	Shareable	Peripherals accessible to all masters.
0xC0000000-0xDFFFFFFF	Device	Device, nGnRE	XN	-	Shareable	Peripherals accessible only to this PE.
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.

a. Write-Through.
b. Read-allocate.
c. Write-back, write allocate.
d. Memory with the Execute Never (XN) memory attribute.

R_{MBRB} An access that crosses a boundary is UNPREDICTABLE. This rule also applies to the `0xFFFFFFFF` - `0x00000000` boundary.

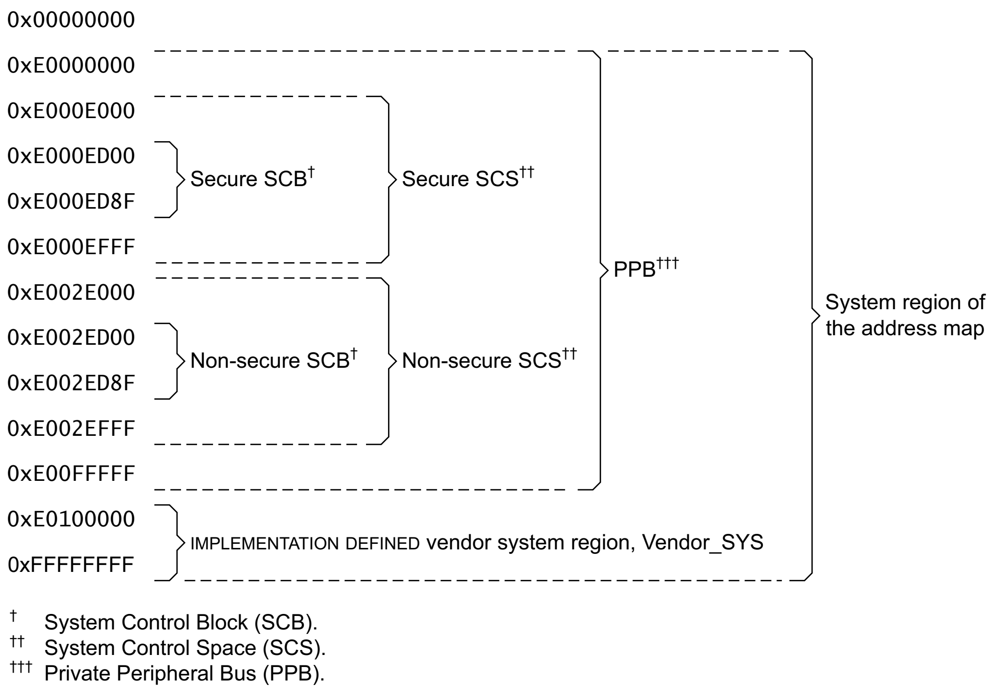
See also:
- *The System region of the system address map* on page B6-190.
- *Address space* on page B5-134.
- *Memory accesses* on page B5-133.
- *Caches* on page B5-172.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FQSD	From 8.0	None	-
MBRB	From 8.0	None	-

B6.2 The System region of the system address map

R_{BDNB} The system region of the system address map is as follows:



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.

I_FWLM Arm recommends that Vendor_SYS is divided as follows:

- `0xE0100000-0xEFFFFFFF` is reserved.
- Vendor resources start at `0xF0000000`.

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.

R_RJHJ If the exception entry context stacking, exception return context unstacking, or lazy floating-point state preservation, results in an access to an address within the PPB space the behavior of the access is CONSTRAINED UNPREDICTABLE and is one of the following:

- Generates a BusFault.
- Perform the specified access to the PPB space.

This does not apply to the `VLSTM` instruction.

See also:

- *System address map* on page B6-188.
- *The System Control Space (SCS)* on page B6-192.
- *STIR, Software Triggered Interrupt Register* on page D1-1141.
- *CCR, Configuration and Control Register* on page D1-888.
- *Debug resources* on page B11-226.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BDNB	From 8.0	None	-
MXRW	From 8.0	M && !S	-
DQQS	From 8.0	None	-
RJHJ	From 8.0	None	-

B6.3 The System Control Space (SCS)

R$_{CQVK}$ The System Control Space (SCS) provides registers for control, configuration, and status reporting.

R$_{CFPK}$ The Secure view of the NS alias is identical to the Non-secure view of normal addresses unless otherwise stated.

R$_{GLNG}$ Privileged accesses to unimplemented registers are RES0.

R$_{NDML}$ Unprivileged accesses to unimplemented registers will generate a BusFault unless otherwise stated.

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.

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.

See also:
- *The System region of the system address map* on page B6-190.
- *System Control Block* on page D1-864.
- *System Control Block (NS alias)* on page D1-869.
- *Debug Control Block* on page D1-866.
- *Debug Control Block (NS alias)* on page D1-871.
- *STIR, Software Triggered Interrupt Register* on page D1-1141.
- *SYST_CSR, SysTick Control and Status Register* on page D1-1144.
- Chapter B10 *Nested Vectored Interrupt Controller*.
- Chapter B8 *The Armv8-M Protected Memory System Architecture*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CQVK	From 8.0	None	-
CFPK	From 8.0	S	-
GLNG	From 8.0	None	-
NDML	From 8.0	M	-
BMLS	From 8.0	None	-
WQQB	From 8.0	None	-

Chapter B7
Synchronization and Semaphores

This chapter specifies the Armv8-M architecture rules for exclusive access instructions and non-blocking synchronization of shared memory. It contains the following sections:

B7.1 Exclusive access instructions

R$_{LQDX}$ Armv8 provides non-blocking synchronization of shared memory, using synchronization primitives for accesses to both Normal and Device memory.

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

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.

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.

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.

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.

See also:

- *The local monitors* on page B7-196.
- *The global monitor* on page B7-198.
- *Exclusive access instructions and the monitors* on page B7-202.
- *Load-Exclusive and Store-Exclusive instruction constraints* on page B7-203.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LQDX	From 8.0	None	-
RGCP	From 8.0	None	-
MWFP	From 8.0	None	-

Rule	Architecture version	Required extensions	Notes
JHMH	From 8.0	None	-
MTTN	From 8.0	None	-
FMXK	From 8.0	None	-

B7.2 The local monitors

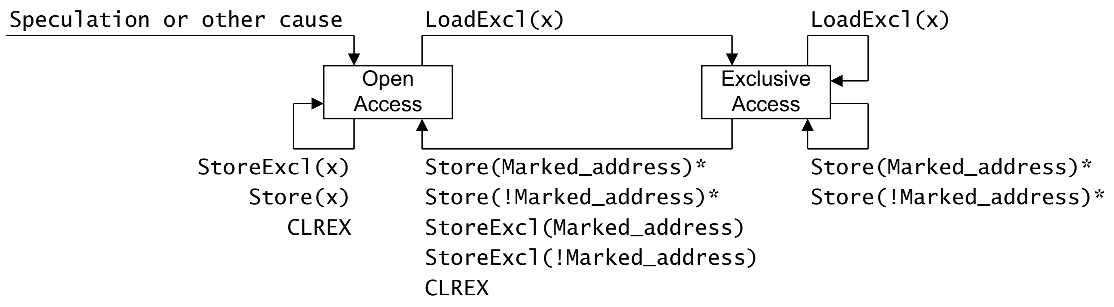
R_{QTFP} Any non-aborted attempt by the same PE to use a Store-Exclusive instruction to modify any address is guaranteed to clear the marking.

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.

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.

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.

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.

R_{JWQS} A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive instructions from other PEs.

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.

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.

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.

R_{HRHC} An exception return clears the local monitor.

See also:

- *Exclusive access instructions and the monitors* on page B7-202.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QTFP	From 8.0	None	-
NJWC	From 8.0	None	-
PFFT	From 8.0	None	-
KXNM	From 8.0	None	-
WTHJ	From 8.0	None	-
JWQS	From 8.0	None	-
KJQW	From 8.0	None	-
XMML	From 8.0	None	-
MRSD	From 8.0	None	-
HRHC	From 8.0	None	-

B7.3 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.

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.

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.

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.

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.

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.

R_{HBKJ} The set of memory types that support atomic instructions includes all of the memory types for which a global monitor is implemented.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FKFB	From 8.0	None	-
VDLP	From 8.0	None	-
NNDC	From 8.0	None	-
HBKJ	From 8.0	None	-
HLHS	From 8.0	None	• AMemManage requires M && MPU • ABusFault requires M
FQRT	From 8.0	None	-

B7.3.1 Load-Exclusive and Store-Exclusive

R_{RXVB} The global monitor only supports a single outstanding exclusive access to shareable memory for each PE.

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.

R_{MPKM} A Load-Exclusive instruction by one PE has no effect on the global monitor state for any other PE.

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.

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.

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.

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.

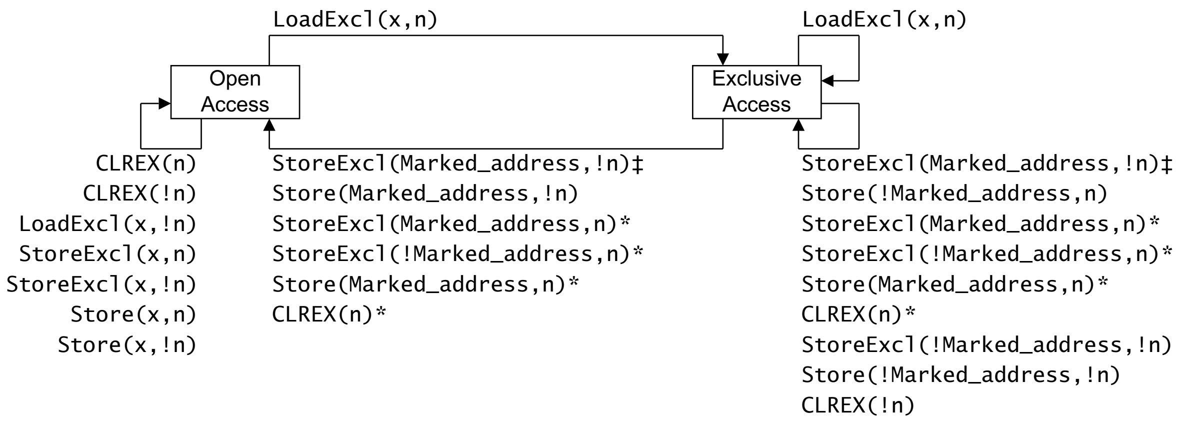
The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RXVB	From 8.0	None	-
GXLF	From 8.0	None	-
MPKM	From 8.0	None	-
MFGC	From 8.0	None	-
NNMG	From 8.0	None	-
WKPJ	From 8.0	None	-
NWWH	From 8.0	None	-

B7.3.2 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.

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.

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.

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.

R_{DLMP} A Load-Exclusive instruction can only update the marked shareable memory address for the PE issuing the Load-Exclusive instruction.

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.

See also:

- *Exclusive access instructions and the monitors* on page B7-202.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HKQT	From 8.0	None	-
GDMD	From 8.0	None	-
RGFK	From 8.0	None	-
QVWF	From 8.0	None	-
DLMP	From 8.0	None	-
BSGB	From 8.0	None	-

B7.4 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.

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.

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.

See also:
- *The local monitors* on page B7-196.
- *The global monitor* on page B7-198.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VXWN	From 8.0	None	-
DTRN	From 8.0	None	-
DFNB	From 8.0	None	-

B7.5 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.

R_{BHPN} The architecture does not require an address or size check as part of the IsExclusiveLocal() function.

R_{LHLG} If two StoreExcl instructions are executed without an intervening LoadExcl instruction the second StoreExcl instruction returns a status value of 1.

R_{DVRQ} The architecture does not require every LoadExcl instruction to have a subsequent StoreExcl instruction.

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.

R_{GVWN} The hardware only ensures that a LoadExcl/StoreExcl pair succeeds if the LoadExcl and the StoreExcl have the same transaction size.

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.

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.

R_{PKQF} The architecture sets an upper limit of 2048 bytes on the Exclusives reservation granule that can be marked as exclusive.

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.

R_{XPDN} After taking a BusFault or a MemManage fault, the state of the exclusive monitors is UNKNOWN.

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.

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.

IMDHL 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.

RRRTJ 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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BHPN	From 8.0	None	-
LHLG	From 8.0	None	-
DVRQ	From 8.0	None	-
JXXS	From 8.0	None	-
GVWN	From 8.0	None	-
PKQF	From 8.0	None	-
XPDN	From 8.0	M	-
FCRN	From 8.0	None	-
DMJW	From 8.0	None	-
RRTJ	From 8.0	None	-

Chapter B8
The Armv8-M Protected Memory System Architecture

This chapter specifies the Armv8-M *Protected Memory System Architecture* (PMSAv-8) rules, and in particular the rules for the optional *Memory Protection Unit* (MPU) and the optional *Security Attribution Unit* (SAU). It contains the following sections:

B8.1 Memory Protection Unit

R_{HPNK} In an implementation that includes the Protected Memory System Architecture (PMSA), system address space is protected by a Memory Protection Unit (MPU).

R_{TBPJ} PMSAv8-M only supports a unified memory model. All enabled regions support instruction and data accesses.

R_{HBNG} Memory attributes are determined from the default system address map or by using an MPU.

R_{BXCN} MPU support in Armv8-M is optional.

R_{MCCL} The default memory map can be configured to provide a background region for privileged accesses.

R_{PBPJ} When the MPU is enabled, the PE can be configured to use the default system map when it processes NMI and HardFault exceptions.

R_{JVJC} When the MPU is disabled or not present, accesses use memory attributes from the default system address map.

R_{KLHL} If the MPU is enabled, attributes for memory accesses that hit in a single region are provided by the hit region.

R_{DBBM} The MPU divides the memory into regions.

R_{JVCN} An individual MPU region is defined by:

```
Address >= MPU_RBAR.BASE:‘00000’ && Address <= MPU_RLAR.LIMIT:‘11111’
```

R_{MNDS} The number of supported MPU regions is IMPLEMENTATION DEFINED.

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.

R_{XGFK} All MPU regions are aligned to a multiple of 32 bytes.

R_{BPGB} The PE can fetch and execute instructions from each MPU region according to the value of MPU_RBAR.XN. The value of MPU_RBAR.XN can be used by privileged software.

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.

R_{ZLHD} Unless otherwise stated all load, store, and instruction fetch transactions are subject to an MPU check.

R_{BDCW} If MPU_CTRL.ENABLE is zero no MPU checks are carried out.

I_{HSCD} The MPU check is one of a number of checks carried out on any load, store or instruction fetch transaction including alignment, and security attribution checks, and a check for any BusFaults.

R_{VHHL} Exception vector reads from the Vector Address Table always use the default system address map and are not subject to an MPU check.

R_{WDNS} Any load, store or instruction fetch transaction where the requested execution priority is negative will not be subject to an MPU check.

R_{TGQD} Any load, store of instruction fetch transaction to the PPB, within the range `0xE0000000-0xE00FFFFF,` is not subject to an MPU check.

R_{BWMB} Unless otherwise stated all load, store or instruction fetch transactions which are subject to an MPU check will also be subject to an MPU region lookup.

R_{QDQS} If MPU_CTRL.ENABLE is zero an MPU region lookup is not carried out.

R_{LLLP} Any MPU 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.

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`. 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.

R_{KMTF} For data accesses, the MPU memory attribution and privilege checking uses the configuration registers that correspond to the current executing Security state of the PE.

R_{RLBR} For instruction fetches, the MPU memory attribution and privilege checking uses the configuration registers associated with the address that is fetched.

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.

See also:

- *System address map* on page B6-188.
- *Access rights* on page B5-142.
- *Device memory attributes* on page B5-159.
- *Shareability attributes* on page B5-165.
- *Memory access restrictions* on page B5-166.
- *Mismatched memory attributes* on page B5-167.
- *Load-Exclusive and Store-Exclusive accesses to Normal memory* on page B5-169.
- *Load-Acquire and Store-Release accesses to memory* on page B5-170.
- *MPU_CTRL, MPU Control Register* on page D1-1086.
- *TT_RESP, Test Target Response Payload* on page D1-1176.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HPNK	From 8.0	MPU	-
TBPJ	From 8.0	MPU	-
HBNG	From 8.0	MPU	-
MCCL	From 8.0	MPU	-
BXCN	From 8.0	MPU	-
PBPJ	From 8.0	MPU	-
JVJC	From 8.0	MPU	-
KLHL	From 8.0	MPU	-
DBBM	From 8.0	MPU	-
JVCN	From 8.0	MPU	-
MNDS	From 8.0	MPU	-
XGFK	From 8.0	MPU	-

Rule	Architecture version	Required extensions	Notes
BPGB	From 8.0	MPU	-
NBPN	From 8.0	MPU	-
BDCW	From 8.0	MPU	-
VHHL	From 8.0	MPU	-
WDNS	From 8.0	MPU	-
TGQD	From 8.0	MPU	-
BWMB	From 8.0	MPU	-
QDQS	From 8.0	MPU	-
LLLP	From 8.0	MPU	-
KDJG	From 8.0	MPU	-
KDJG	From 8.0	MPU	-
KMTF	From 8.0	MPU && S	-
RLBR	From 8.0	MPU	-
PLJG	From 8.0	MPU	-

B8.2 Security attribution

I_{SBSJ} The Secure Attribution Unit and the Implementation Defined Attribution Unit are collectively referred to as the Attribution Unit (AU).

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.

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.

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.

R_{NJGR} An *attribution unit* (AU) violation is defined as being a violation raised by either the SAU or the IDAU.

R_{QGVS} All boundaries between address ranges with different security attributes are aligned to 32-byte boundaries.

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.

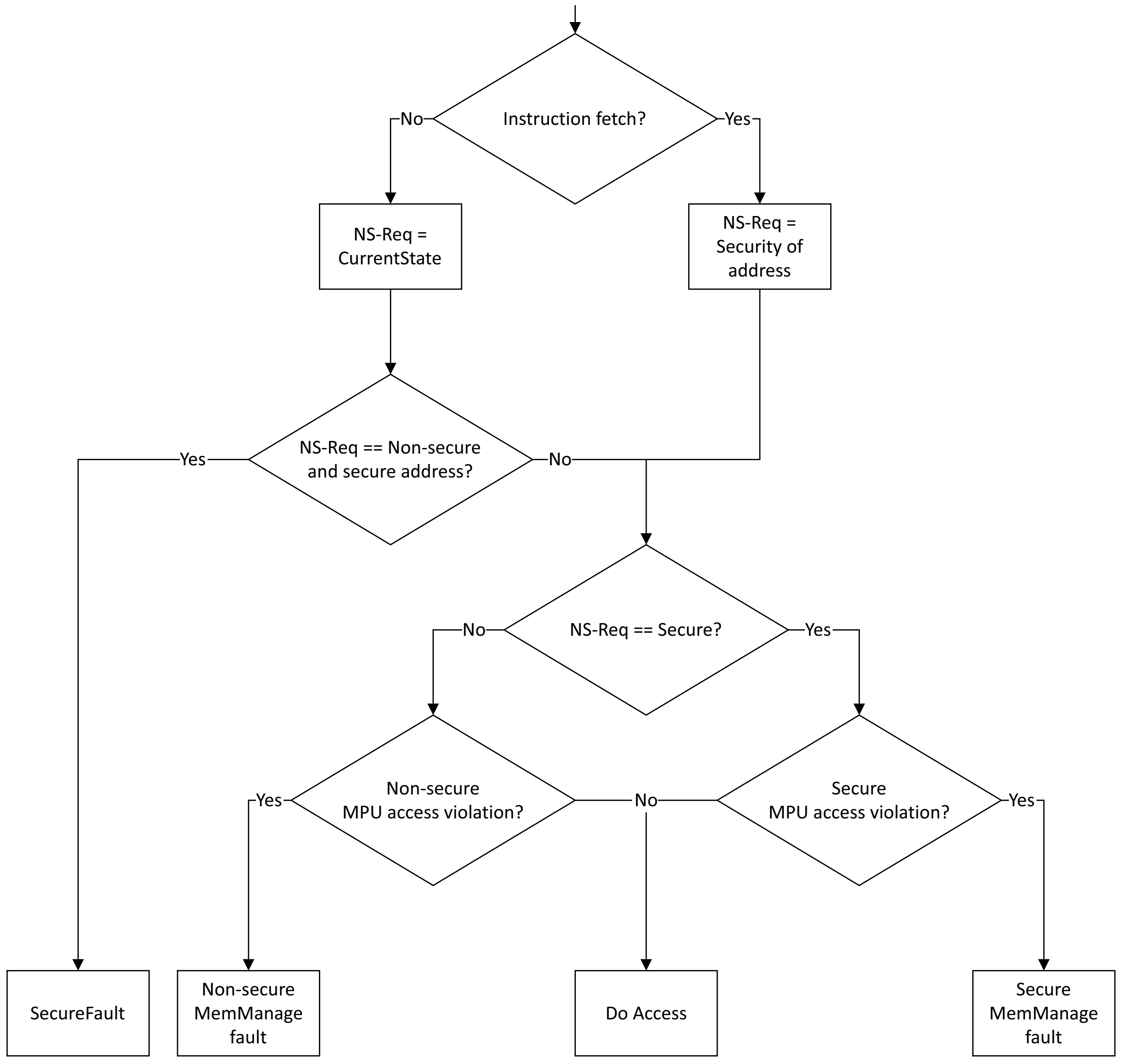
`0xE0040000` - `0xE0041FFF`: TPIU, ETM.

`0xE00FF000` - `0xE00FFFFF`: ROM table.

`0xE0000000` - `0xEFFFFFFF` for instruction fetch only.

Additional address ranges specified by the IDAU.

IVPWL The Security attribution and MPU check sequence is shown in the following diagram.


Instruction fetch?
No
Yes
NS-Req = CurrentState
NS-Req = Security of address
NS-Req == Non-secure and secure address?
Yes
No
NS-Req == Secure?
No
Yes
Non-secure MPU access violation?
Yes
No
Secure MPU access violation?
Yes
SecureFault
Non-secure MemManage fault
Do Access
Secure MemManage fault

See also:

- *Security attribution unit (SAU)* on page B8-212.
- *IMPLEMENTATION DEFINED Attribution Unit (IDAU)* on page B8-214.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JGHS	From 8.0	S	-
RPKG	From 8.0	S	-
MGXN	From 8.0	S	-

Rule	Architecture version	Required extensions	Notes
NJGR	From 8.0	S	-
QGVS	From 8.0	S	-
BLJT	From 8.0	None	• Some ranges require S • Some ranges require DB

B8.3 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.

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).

R_{RVFP} The SAU region configuration fields can only be accessed indirectly using the window registers 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

R_{NBFD} When the SAU is enabled, an address is defined as matching a region in the SAU if the following is true:

```
SAU_REGIONn.BADDR <= Address <= SAU_REGIONn.LADDR.
```

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.

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.

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.

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.

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 specified in the unimplemented region returns an UNKNOWN value.
- Any write to a register specified in the unimplemented region becomes UNKNOWN.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
VFLR	From 8.0	S	-
RVFP	From 8.0	S	-
NBFD	From 8.0	S	-
MPJC	From 8.0	S	-
WGDK	From 8.0	S	-

Rule	Architecture version	Required extensions	Notes
GVFQ	From 8.0	S	-
MBJN	From 8.0	S	-
BBCT	From 8.0	S	-

B8.4 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.

R_{MBHQ} The Non-secure and the Non-secure callable attributes from the IDAU are combined with the results from the SAU. The resulting attribution is the most restrictive of the two.

See also:

- *TT, TTT, TTA, TTAT* on page C2-720.
- *Security attribution* on page B8-209.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MVCM	From 8.0	S	-
MBHQ	From 8.0	S	-

Chapter B9
The System Timer, SysTick

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

B9.1 The system timer, SysTick

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.

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.

R_{CNTG} In a PE with the Main Extension but without the Security Extension, one system timer, SysTick, is implemented.

R_{XPCW} In a PE with the Main and Security Extensions, two system timers are implemented:

- SysTick, Secure instance.
- SysTick, Non-secure instance.

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.

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. 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.

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.

R_{TTFT} A write to SYST_CVR does not cause a SysTick exception.

I_{VDJQ} Setting SYST_CSR.TICKINT to 1 causes the SysTick exception to become pending on the SysTick reaching zero.

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.

I_{MMRQ} Software can optionally use SYST_CALIB.TENMS to scale the counter to other clock rates within the dynamic range of the counter.

R_{QSKV} When the PE is halted in Debug state, any implemented SysTicks do not decrement.

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, jitter.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BQRG	From 8.0	!M && !S && ST	-
PDDL	From 8.0	!M && S && ST	-
CNTG	From 8.0	M && !S && ST	-
XPCW	From 8.0	M && S && ST	-
TLGK	From 8.0	S && ST	-
TTFT	From 8.0	ST	-
QSKV	From 8.0	ST	-

Chapter B10
Nested Vectored Interrupt Controller

This chapter specifies the Armv8-M *Nested Vectored Interrupt Controller* (NVIC) rules. It contains the following sections:

B10.1 NVIC definition

R$_{XJJQ}$ An Armv8-M PE includes an integral interrupt controller.

R$_{WQHG}$ The Interrupt Controller Type Register (ICTR) defines the number of external interrupt lines that are supported.

See also:

- *ICTR, Interrupt Controller Type Register* on page D1-1033.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XJJQ	From 8.0	None	-
WQHG	From 8.0	None	-

B10.2 NVIC operation

R_{SNVK} It is IMPLEMENTATION DEFINED which NVIC interrupts are implemented.

R$_{SGCR}$ When a particular NVIC interrupt line is not implemented, the registers that are associated with it are reserved.

R$_{CCVJ}$ Only an interrupt that is both pending and enabled can preempt PE execution.

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.

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.

I$_{HVQQ}$ For some implementations, pulse-sensitive interrupt signals are held long enough to ensure that the PE can sample them reliably.

R$_{QKFW}$ All NVIC interrupts have a programmable priority value and an associated exception number.

R$_{XNQW}$ NVIC interrupts can be enabled and disabled by writing to their corresponding Interrupt Set-Enable or Interrupt Clear-Enable register bit field.

R$_{WGDJ}$ An implementation can hard-wire interrupt enable bits to zero if the associated interrupt line does not exist.

R$_{RSDJ}$ An implementation can hard-wire interrupt enable bits to one if the associated interrupt line cannot be disabled.

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.

See also:

- *Exception numbers and exception priority numbers* on page B3-53.
- *Priority model* on page B3-66.
- *Nested Vectored Interrupt Controller* on page D1-864.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
SNVK	From 8.0	None	-
SGCR	From 8.0	None	-
CCVJ	From 8.0	None	-
CVJS	From 8.0	None	-
QKFW	From 8.0	None	-

Rule	Architecture version	Required extensions	Notes
XNQW	From 8.0	None	-
WGDJ	From 8.0	None	-
RSDJ	From 8.0	None	-
NRJV	From 8.0	None	-

Chapter B11
Debug

This chapter specifies the Armv8-M debug rules. It contains the following sections:

B11.1 Debug feature overview

R_{WXRJ} The debug configuration of an implementation is IMPLEMENTATION DEFINED.

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[a]	Optional	Requires Halting debug
FPB[b]	Optional	Requires Halting debug
DWT-D[c]	Optional	Requires Halting debug
DWT-T[d]	Requires ITM and DWT-D	Never implemented
ITM[e]	Optional	Never implemented
CTI[f]	Requires ETM or Halting debug	Requires ETM or Halting debug
TPIU[g]	Requires ITM or ETM	Requires ETM
ETM[h]	Optional	Optional

a. External Debug Request
b. Flash Patch and Breakpoint Unit
c. Data, Watchpoint and Trace Unit - debug functionality
d. Data Watchpoint and Trace Unit - trace functionality
e. Instrumentation Trace Macrocell
f. Cross Trigger Interface
g. Trace Port Interface Unit
h. Embedded Trace Macrocell

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).

I_{SFSG} The recommended debug implementation levels are:

Level	With Main Extension	Without Main Extension
Minimum	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.	No debug support. DFSR, DHCSR, DCRSR, DCRDR, and DEMCR are RES0. ID_DFR0 is RAZ.
Basic	Adds support for Halting debug with: • 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. This support is identified in ID_DFR0.	Adds support for Halting debug with: • Debug Access Port and ROM table. • SHCSR, DFSR, DHCSR, DCRSR, DCRDR, and DEMCR are implemented. Access from the PE is IMPLEMENTATION DEFINED. • FPB with at least two breakpoints. • DWT with at least: — One watchpoint, that supports instruction and data address matching. — DWT_PCSR. • Optional support for a CTI in a multiprocessor system. This support is identified in ID_DFR0.
Comprehensive	Adds basic trace support with: • ITM. • DWT with: — Trace support. — Profiling support. — Cycle counter. • TPIU.	Not applicable without the Main Extension.
Program trace	Adds ETM.	Adds ETM and TPIU.

See also:
- *Debug mechanisms* on page B11-226.
- *Halting debug* on page B11-243.
- *DebugMonitor exception* on page B11-245.
- *Breakpoint instructions* on page B11-252.
- *Instrumentation Trace Macrocell* on page B12-260.
- *Data Watchpoint and Trace unit* on page B12-270.
- *Embedded Trace Macrocell* on page B12-291.
- *Trace Port Interface Unit* on page B12-293.
- *Flash Patch and Breakpoint unit* on page B12-295.
- *DEMCR, Debug Exception and Monitor Control Register* on page D1-928.
- *DFSR, Debug Fault Status Register* on page D1-933.
- *DHCSR, Debug Halting Control and Status Register* on page D1-935.

- *DCRDR, Debug Core Register Data Register* on page D1-922.
- *DCRSR, Debug Core Register Select Register* on page D1-923.
- *ID_DFR0, Debug Feature Register 0* on page D1-1035.
- *DWT_PCSR, DWT Program Counter Sample Register* on page D1-978.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WXRJ	From 8.0	DB	-
FMQF	From 8.0	DB	-
FHRN	From 8.0	-	CTI requires Halting debug or ETM

B11.1.1 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.

I_{LBLF} When the PE is halted, it is in *Debug state*.

I_{SXVR} When the PE is not halted, it is in *Non-debug state*.

See also:

- *Accessing debug features* on page B11-229.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HWCH	From 8.0	DB	-

B11.1.2 Debug resources

R_{TZVG} In the system address map, debug resources are in the *Private Peripheral Bus* (PPB) region.

R_{FBHD} Except for the resources in the SCS, each debug component occupies a fixed 4KB address region.

R_{WXTK} The debug resources in the SCS are:

- The *Debug Control Block* (DCB).

- Debug controls in the *System Control Block* (SCB).

IKKBT If the Main Extension is implemented, then support for DebugMonitor is implemented. If the Main Extension is not implemented, then DebugMonitor is not supported.

RVMGD ROM table entries identify which optional debug components are implemented.

RRNXK 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`	*Flash Patch and Breakpoint* (FPB) unit
`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

See also:
- *Instrumentation Trace Macrocell* on page B12-260.
- *Data Watchpoint and Trace unit* on page B12-270.
- *Flash Patch and Breakpoint unit* on page B12-295.
- Chapter B6 *The System Address Map*.
- *Debug System registers* on page B11-231.
- *Trace Port Interface Unit* on page B12-293.
- *Embedded Trace Macrocell* on page B12-291.
- *ROM table* on page B11-229.
- *Accessing debug features* on page B11-229.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
TZVG	From 8.0	None	-
FBHD	From 8.0	DB	-
WXTK	From 8.0	DB	-
VMGD	From 8.0	DB	-
RNXK	From 8.0	DB	-

B11.1.3 Trace

R_{LJVL} Trace can be generated by using the:

- *Embedded Trace Macrocell* (ETM).
- *Instrumentation Trace Macrocell* (ITM).
- *Data Watchpoint and Trace* (DWT) unit.

R_{NFVB} A debug implementation that generates trace includes a trace sink, such as a TPIU.

I_{RJKJ} A TPIU can be either the Armv8-M TPIU implementation, or an external system resource.

See also:

- Chapter F1 *ITM and DWT Packet Protocol Specification*.
- The applicable ETM Architecture Specification.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LJVL	From 8.0	ETM || ITM || DWT-T	-
NFVB	From 8.0	(ETM || ITM || DWT-T) && TPIU	-

B11.2 Accessing debug features

R_{WVSZ} The mechanism by which an external debugger accesses the PE and system is IMPLEMENTATION DEFINED.

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.

R_{WPGQ} Writes from a DAP are complete when the DAP reports them as complete.

R_{WCQK} For SCS registers, a write from a DAP is complete when the write has completed and the SCS register has been updated.

R_{JRHS} Software configures and controls the debug model through memory-mapped registers.

See also:
- *ROM table*.
- *DAP access permissions* on page B11-237.
- The *ARM® Debug Interface v5 Architecture Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WVSZ	From 8.0	DB	-
WPGQ	From 8.0	DB	-
WCQK	From 8.0	DB	-
JRHS	From 8.0	DB	-

B11.2.1 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.

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.

I_{PHJJ} For an Armv8-M ROM table, all address offsets are negative.

R_{GPPX} The ROM table is implemented if any other debug component is implemented or a Debug Access Port is implemented.

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.

R_{NDQW} If a debug component is implemented, debug registers can provide additional information about the implemented features of that debug component.

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[a]	Points to the Instrumentation Trace unit at `0xE0000000`.
`0x010`	`0xFFF41002` or `0xFFF41003`	ROMTPIU[b]	Points to the Trace Port Interface Unit.
`0x014`	`0xFFF42002` or `0xFFF42003`	ROMETM[b]	Points to the Embedded Trace Macrocell unit.
-	`0x00000000`	End	End-of-table marker. It is IMPLEMENTATION DEFINED whether the table is extended with pointers to other system debug resources. The table entries always terminate with a null entry.
`0x020-0xEFC`	-	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 CoreSight 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	

a. Accesses cannot cause a non-existent memory exception.
b. It is IMPLEMENTATION DEFINED whether a shared resource is managed by the local PE or a different resource.

R_{RGVM} The entry `0x00000000` is the end-of-table marker.

See also:

- *CoreSight and identification registers*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GPPX	From 8.0	DB	-
BQSP	From 8.0	DB	-
NDQW	From 8.0	DB	-
DPVG	From 8.0	DB	Requires the extensions indicated in the rule.

B11.2.2 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.

See also:

- Chapter D1 *Register Specification*.
- *Debug Control Block* on page D1-866.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RHDW	From 8.0	DB	-

B11.2.3 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.

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`

Debug component	Space reserved for ID registers	Space reserved for CoreSight compliance
SCS	`0xE000EFD0-0xE000EFFC`	`0xE000EFA0-0xE000EFCC`
TPIU	`0xE0040FD0-0xE0040FFC`	`0xE0040FA0-0xE0040FCC`
ETM	`0xE0041FD0-0xE0041FFC`	`0xE0041FA0-0xE0041FCC`
ROM table	`0xE00FFFD0-0xE00FFFFC`	`0xE00FFFA0-0xE00FFFCC`

R_{VWSX} For the ROM table, the ID register space is used for a set of CoreSight-compliant ID registers.

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.

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.

I_{HQSR} The Part number in the PIDR registers should be assigned a unique value for each implementation, 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 DEVTYPE or DEVARCH registers.

I_{CTBF} The Part number in the PIDR registers do not need to be unique for different implementation options of the same part.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CBCM	From 8.0	DB	-
VWSX	From 8.0	DB	-
HXDK	From 8.0	DB	-
VQPM	From 8.0	DB	-

B11.3 Debug authentication interface

I_{GWTN} The following pseudocode functions provide an abstracted description of the authentication interface:

- `ExternalInvasiveDebugEnabled()`.
- `ExternalSecureInvasiveDebugEnabled()`.
- `ExternalNoninvasiveDebugEnabled()`.
- `ExternalSecureNoninvasiveDebugEnabled()`.

I_{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.

R_{HVGN} For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is FALSE, then `ExternalSecureInvasiveDebugEnabled()` is FALSE.

R_{JWCS} For any implementation of the authentication interface, if `ExternalNoninvasiveDebugEnabled()` is FALSE, then `ExternalSecureNoninvasiveDebugEnabled()` is FALSE.

R_{XCMD} For any implementation of the authentication interface, if `ExternalInvasiveDebugEnabled()` is TRUE, then `ExternalNoninvasiveDebugEnabled()` is TRUE.

R_{LCHH} For any implementation of the authentication interface, if `ExternalSecureInvasiveDebugEnabled()` is TRUE, then `ExternalSecureNoninvasiveDebugEnabled()` is TRUE.

I_{MSRG} Secure self-hosted debug is controlled by the authentication interface. The pseudocode function `ExternalSecureSelfHostedDebugEnabled()` provides an abstracted description of this authentication interface.

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.

See also:

- *Halting debug authentication* on page B11-234.
- *DebugMonitor exception authentication* on page B11-235.
- *Non-invasive debug authentication* on page B11-236.
- *DAP access permissions* on page B11-237.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HVGN	From 8.0	DB	-
JWCS	From 8.0	DB	-
XCMD	From 8.0	DB	-
LCHH	From 8.0	DB	-
GLWM	From 8.0	DB	-

B11.3.1 Halting debug authentication

IDMFG 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()`.

RJJJK Unless otherwise stated Halting is prohibited in all states if the function `ExternalInvasiveDebugEnabled()` returns FALSE.

RJXTX When the PE is halted, the PE behaves as if `ExternalInvasiveDebugEnabled()` is TRUE. The pseudocode function `HaltingDebugAllowed()` describes this.

IBCZM 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
	1	Halting is allowed
TRUE	X	

RBMRJ 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.

RKBKM If 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.

RKMXG If the PE is in Debug state:

- DHCSR.S_SDE reads as one if any one of the following of 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.

ILDTR 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. Halting is prohibited in Secure state.
	1	Halting is allowed.

RFXCB When DHCSR.C_DEBUGEN == 0 or the PE is in a state in which halting is prohibited, the PE does not enter Debug state.

See also:

- `CanHaltOnEvent()`.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JJJK	From 8.0	Halting debug	-
JXTX	From 8.0	Halting debug	-
BMRJ	From 8.0	Halting debug && S	-
KBKM	From 8.0	Halting debug	S is required for Secure behavior.
KMXG	From 8.0	Halting debug	S is required for Secure behavior.
FXCB	From 8.0	Halting debug	S is required for Secure behavior.

B11.3.2 DebugMonitor exception authentication

R_{MXTM}

R_{MXTM} DebugMonitor exception authentication is only available if the Main Extension is implemented.

R_{LQCN} DebugMonitor exception authentication is controlled by the IMPLEMENTATION DEFINED authentication interface function `ExternalSecureSelfHostedDebugEnabled()`.

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.

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.

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.

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

R_{YFPK} If DEMCR.SDME == 1, SHPR3.PRI_12 behaves as RES0 when accessed from Non-secure state.

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.

See also:

- CanPendMonitorOnEvent().

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MXTM	From 8.0	M	-
LQCN	From 8.0	M && S	-
QJQN	From 8.0	M && S	-
CPPN	From 8.0	M	-
WXMG	From 8.0	M	-
YFPK	From 8.0	M && S	-
HXLX	From 8.0	M	-

B11.3.3 Non-invasive debug authentication

R_{GFTG} Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function ExternalNoninvasiveDebugEnabled().

R_{HXQD} Non-invasive authentication is controlled by the IMPLEMENTATION DEFINED function ExternalSecureNoninvasiveDebugEnabled().

R_{CFNB} When HaltingDebugAllowed() is TRUE, the PE behaves as if ExternalNoninvasiveDebugEnabled() returns TRUE. The pseudocode function NoninvasiveDebugAllowed() describes this.

R_{QMRF} Non-invasive debug is prohibited if the function NoninvasiveDebugAllowed() returns FALSE.

I_{PHPR} If the Security Extension is not implemented, there are two non-invasive debug authentication modes:

ExternalNoninvasiveDebugEnabled()	HaltingDebugAllowed()	Non-invasive debug authentication mode
FALSE	FALSE	Non-invasive debug prohibited.
	TRUE	Non-invasive debug allowed.
TRUE	X	

R_{MLPS} Non-invasive debug of Secure operations is prohibited if any of:

- NoninvasiveDebugAllowed() returns FALSE.
- DHCSR.S_SDE is set to 0, DAUTHCTRL.SPNIDENSEL is set to 1 and DAUTHCTRL.INTSPNIDEN is set to 0.
- DHCSR.S_SDE is set to 0, DAUTHCTRL.SPNIDENSEL is set to 0 and ExternalSecureNoninvasiveDebugEnabled() returns FALSE.

The pseudocode function `SecureNoninvasiveDebugAllowed()` shows this, if this function returns true Secure Non-invasive debug is permitted.

I_{PNRC} If the Security Extension is implemented, there are three non-invasive debug authentication modes:

`NoninvasiveDebugAllowed()`	`SecureNoninvasiveDebugAllowed()`	**Non-invasive debug authentication mode**
FALSE	X	Non-invasive debug prohibited.
TRUE	FALSE	Non-invasive debug of only Non-secure operations allowed. Non-invasive debug of Secure operations prohibited.
	TRUE	Non-invasive debug allowed.

RLXRK The PE does not generate any trace or profiling data when non-invasive debug is prohibited.

RVYGT If non-invasive debug of Secure operations is prohibited, the PE does not generate any trace or profiling data that contains secure information.

RTWDH If non-invasive debug is prohibited in the current Security state, an ETM behaves as described in the relevant ETM architecture.

See also:
- `NoninvasiveDebugAllowed()`.
- `SecureNoninvasiveDebugAllowed()`.
- *DWT unit operation* on page B12-271.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GFTG	From 8.0	DB	-
HXQD	From 8.0	DB	-
CFNB	From 8.0	DB	-
QMRF	From 8.0	DB	-
MLPS	From 8.0	DB && S	-
LXRK	From 8.0	DB	-
VYGT	From 8.0	DB && S	-
TWDH	From 8.0	DB && S && ETM	-

B11.3.4 DAP access permissions

RBFSB When `HaltingDebugAllowed()` returns TRUE the external debugger can access the whole physical address space.

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[a]	CoreSight Software Lock registers	No access	RW
0xE00xxFD0-0xE00xxFFF[b]	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	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED	IMPLEMENTATION DEFINED
0xE00FF000-0xE00FFFFF	ROM table	RO	RO
All other PPB regions and registers		No access	No access
0xE0100000-0xFFFFFFFF	Vendor_SYS	No access	RW

a. For each debug component implementing the CoreSight Software Lock registers. These registers are optional.
b. For each debug component implementing the CoreSight ID registers. These registers are optional.

R$_{FFPN}$ The DAP is capable of requesting Secure and Non-secure accesses.

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.

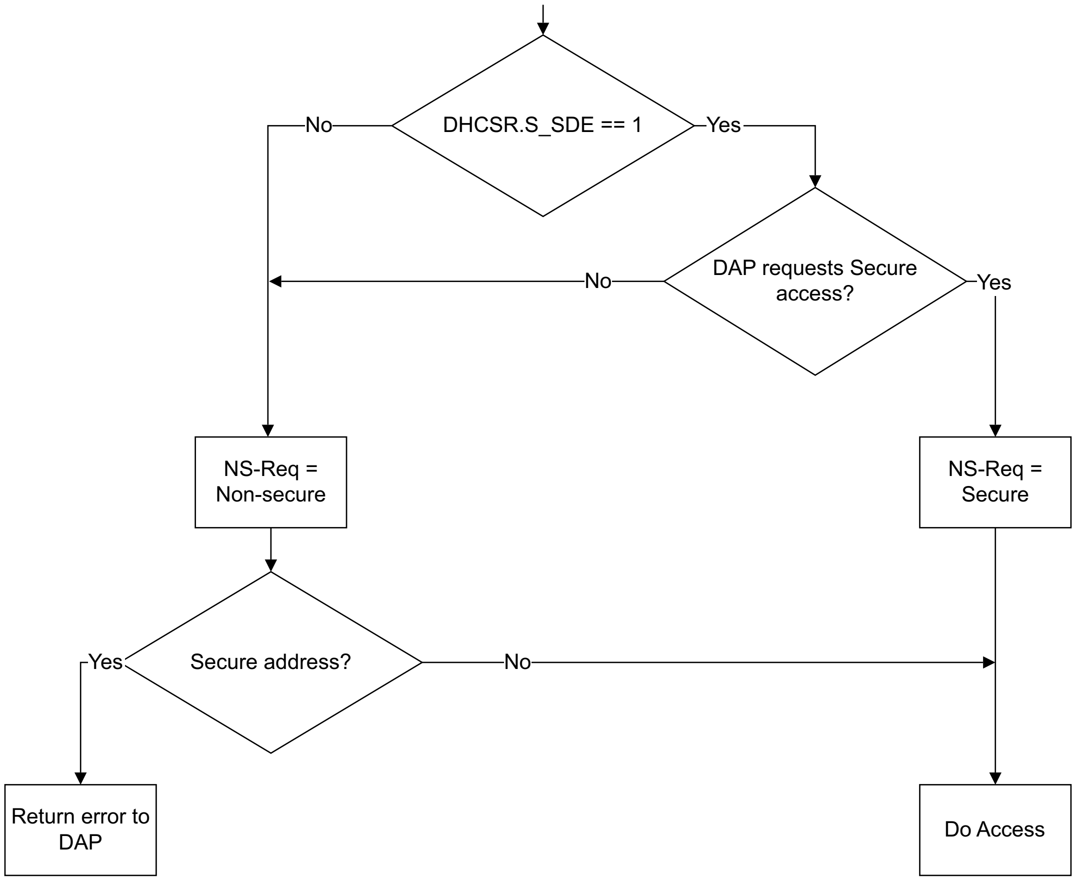
- CSW[23], CSW.SPIDEN, is Read-As-One. This is because the DAP can always request a Secure access.

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 should be tied HIGH.

R$_{JHBC}$ If DHCSR.S_SDE == 1, and the DAP requests a Secure access, NS-Req is set to Secure.

R$_{LVBG}$ If either DHCSR.S_SDE == 0 or the DAP requests a Non-secure access, NS-Req set to Non-secure.

R$_{\text{WMRR}}$ 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.



R$_{\text{VTTN}}$ DAP accesses are not checked by the MPU.

R$_{\text{FDCQ}}$ DAP accesses to the SCS registers ignore NS-Req.

R$_{\text{SSVN}}$ 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	X	Secure
	0	X	Non-secure	Non-secure
		X	Secure	Secure

R$_{\text{HXMG}}$ 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.

See also:

- *Secure address protection* on page B3-70.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BFSB	From 8.0	DB	-
DVSN	From 8.0	DB	-
FFPN	From 8.0	DB && S	-
JHBC	From 8.0	DB && S	-
LVBG	From 8.0	DB && S	-
WMRR	From 8.0	DB && S	-
VTTN	From 8.0	DB && MPU	-
FDCQ	From 8.0	DB && S	-
SSVN	From 8.0	DB && S	-
HXMG	From 8.0	DB	-

B11.4 Debug event behavior

B11.4.1 About debug events

I~CBWT~ An event that is triggered for debug reasons is known as *a debug event*.

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.
- An unrecoverable error.

R~QLTQ~ A debug event that is not ignored, can cause a DebugMonitor exception to occur.

R~MNKP~ The HardFault exceptions or unrecoverable errors 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.

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

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

R~HNRV~ It is IMPLEMENTATION DEFINED whether the DFSR debug event bits are updated when an event is ignored.

I~NSMV~ Debug events are either synchronous or asynchronous.

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.

R_{PVGM} A single instruction can generate several synchronous debug events.

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.

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.

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.C_HALT is set by DHCSR.C_STEP of the previous instruction.
2. Highest-priority pending exception that is eligible to be taken. If the Main Extension is implemented, this might be a DebugMonitor exception, if DEMCR.MON_PEND == 1. This includes where DEMCR.MON_PEND is set by DEMCR.MON_STEP of the previous instruction.
3. Vector catch.
4. Fault from an instruction fetch, including synchronous BusFault error.
5. Breakpoint that is signaled by an FPB unit.
6. `BKPT` instruction or other exception that results from decoding the instructions. This includes the cases where exceptions from the instruction are UNDEFINED, an unimplemented or disabled coprocessor is targeted, or the EPSR.T bit has a value of 1.
7. Other synchronous exception that is generated by executing the instruction, including an exception that is generated by a memory access that is generated by the instruction.

R_{BQVF} The highest-priority synchronous debug event is reported in the DFSR.

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.

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.

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 the PE behaves as if these bits are 0.

See also:

- *Priority model* on page B3-66.
- *Halting debug* on page B11-243.
- *DebugMonitor exception* on page B11-245.
- *Vector catch* on page B11-249.
- `BKPTInstrDebugEvent()`.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PQKW	From 8.0	DB	• Entry to Debug state requires Halting debug
QLTQ	From 8.0	M	-
MNKP	From 8.0	M	An FPB requires FPB
WCPW	From 8.0	DB	• A DebugMonitor exception requires M • Halt requires Halting debug
LDRZ	From 8.0	M || Halting debug	-
HNRV	From 8.0	DB	-
VSVN	From 8.0	DB	-
PVGM	From 8.0	DB	-
WJFB	From 8.0	DB	-
RNRD	From 8.0	DB	-
JSPS	From 8.0	DB	Not all of the Debug features listed might be implemented in a particular implementation
BQVF	From 8.0	DB	-
FWQQ	From 8.0	DB	-
TKRS	From 8.0	DB	-
MRMC	From 8.0	DB	-

Halting debug

R_{WLCF} Setting the DHCSR.C_DEBUGEN bit to 1 enables 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.

R_{THLS} If DHCSR.C_HALT has a value of 1 and halting is allowed, the PE halts and enters Debug state.

R_{FKWB} A debug event that sets DHCSR.C_HALT to 1 pends entry to Debug state.

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.

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.

R$_{JXQF}$ DFSR is updated at the same time as the PE sets DHCSR.C_HALT to 1.

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.

R$_{FRJC}$ For asynchronous debug events, if halting is allowed, the PE enters Debug state in finite time.

R$_{VJKX}$ Entering Debug state has no architecturally defined effect on the Event Register and exclusive monitors.

I$_{JNGH}$ DHCSR.C_SNAPSTALL may allow imprecise entry into the Debug state, for example by forcing any stalled load or store instructions to be abandoned.

R$_{BTBJ}$ If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, DHCSR.C_SNAPSTALL is ignored and the PE behaves as if this bit is 0.

R$_{HLNF}$ If DHCSR.S_SDE == 0, DHCSR.C_SNAPSTALL ignores writes from the debugger.

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.

See also:
- *DHCSR, Debug Halting Control and Status Register* on page D1-935.
- *Debug stepping* on page B11-246.
- *Debug state* on page B11-254.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WLCF	From 8.0	Halting debug	-
RZTG	From 8.0	Halting debug	-
THLS	From 8.0	Halting debug	-
FKWB	From 8.0	Halting debug	-
MXLF	From 8.0	Halting debug	-
XSRJ	From 8.0	Halting debug || EDBGRQ	-
JXQF	From 8.0	Halting debug	-
TXWB	From 8.0	Halting debug	-
VJKX	From 8.0	Halting debug	-
BTBJ	From 8.0	Halting debug	-
HLNF	From 8.0	Halting debug && S	-
RKBK	From 8.0	Halting debug	-

DebugMonitor exception

I_{DPCC} The DebugMonitor exception is only available if the Main Extension is implemented.

R_{ZBSJ} The DebugMonitor exception is enabled when the DEMCR.MON_EN bit is set to 1.

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.

R_{XNMW} If a Debug event does not generate an entry to Debug state and either DEMCR.MON_EN is set to 0 or the DebugMonitor exception group priority is greater 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.

R_{CHXQ} A debug event that sets DEMCR.MON_PEND to 1 pends a DebugMonitor exception.

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.

R_{BRXT} DFSR is updated at the same time as the PE sets DEMCR.MON_PEND to 1.

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.

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.

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.

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.

R_{LNCJ} Asynchronous DebugMonitor exceptions can only cause preemption at instruction boundaries.

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.

See also:

- *DWT unit operation* on page B12-271.
- *FPB unit operation* on page B12-295.
- *Exceptions, instruction resume, or instruction restart* on page B3-95.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
ZBSJ	From 8.0	M	-
PPLF	From 8.0	M	-
XNMW	From 8.0	M	-
CHXQ	From 8.0	M	-
VSPX	From 8.0	M	-
BRXT	From 8.0	M	-
BKHP	From 8.0	M	-
VFLQ	From 8.0	M	-
JVSC	From 8.0	M	-
XPBN	From 8.0	M	-
LNCJ	From 8.0	M	-

B11.4.2 Debug stepping

R_{HMCN} The Armv8-M architecture supports debug stepping in both Halting debug and for 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.

R_{LLVC} If a debug event wakes a `WFE` or `WFI` instruction, then on taking the debug event, the instruction has retired.

See also:

- *Halting debug stepping* on page B11-247.
- *Debug monitor stepping* on page B11-248.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HMCN	From 8.0	Halting debug || M	Might require the DebugMonitor exception
THTG	From 8.0	Halting debug || M	-
LLVC	From 8.0	Halting debug || M	-

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.

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.

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.

I_{LTRX} The debugger can optionally set the DHCSR.C_MASKINTS bit to 1 to prevent PendSV, SysTick, and external configurable interrupts from being taken. When DHCSR.C_MASKINTS is set to 1, 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.

R_{ZDYR} If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, DHCSR.C_MASKINTS is ignored and the PE behaves as if this bit is 0.

R_{FWSN} If DHCSR.S_SDE == 0, DHCSR.C_MASKINTS is ignored for exceptions targeting Secure state.

RMBCB DHCSR.{C_HALT, C_STEP, C_MASKINTS} can be written in a single write to DHCSR, as follows:

DHCSR write[a]			**Effect**
C_HALT	**C_STEP**	**C_MASKINTS**	
0	0	0	Exit Debug state and start instruction execution. Exceptions can become active[b].
0	0	1	Exit Debug state and start instruction execution. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active[b].
0	1	0	Exit Debug state, step an instruction and halt. Exceptions can become active[b].
0	1	1	Exit Debug state, step an instruction and halt. PendSV, SysTick and, external configurable interrupts are disabled, otherwise exceptions can become active[b].
1	X	X	Remain in Debug state.

a. Assumes DHCSR.C_DEBUGEN and DHCSR.S_HALT are both set to 1 when the write occurs, meaning the PE is halted.
b. That is, exceptions become active, based on their configuration, according to the exception priority rules.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
SWKC	From 8.0	Halting debug	-
ZVKS	From 8.0	Halting debug	-
ZDYR	From 8.0	Halting debug	-
FWSN	From 8.0	Halting debug && S	-
MBCB	From 8.0	Halting debug	-

Debug monitor stepping

IDXCT 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.

RMLRM 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 instruction was executed in Non-secure state or the exception was taken from Non-secure state.
- Execution priority is below the priority of the DebugMonitor exception when the instruction was executed or below the exception taken.

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.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MLRM	From 8.0	M	-
MWFT	From 8.0	M	-

B11.4.3 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.

R$_{JCXR}$ Vector catching is only supported by 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.

IXDGP Late arrival and derived exceptions might occur, preempting the exception targeted by the vector catch and postponing when the PE halts.

ILTSX The exception, fault status bit, and vector catch bit are described in the following table.

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

Exception	Fault status bit	Vector catch bit DEMCR
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	-	-

RLKNL 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.

RWRMQ If debug is enabled, DHCSR.C_DEBUGEN == 1, and DEMCR.VC_CORERESET == 1 when the PE resets, the PE pends a Vector Catch debug event, debug is prohibited in Secure state, and the PE has reset into Secure state. The PE does not halt until either it enters Non-secure state or debug is allowed in Secure state.

See also:
- *Exception enable, pending, and active bits* on page B3-57
- *Priority model* on page B3-66.
- *Faults* on page B3-62.
- *Exception numbers and exception priority numbers* on page B3-53.
- *Exceptions during exception entry* on page B3-91.
- *Exceptions during exception return* on page B3-92.
- *Resets, Cold reset, and Warm reset* on page B1-32.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JCXR	From 8.0	Halting debug	-
PBVX	From 8.0	Halting debug	If the Main Extension is not implemented only bits [24],[10] and [0] of DEMCR are implemented with Halting debug functionality. SecureFault requires S.
LKNL	From 8.0	Halting debug && S	-
WRMQ	From 8.0	Halting debug && S	-

B11.4.4 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.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CRJG	From 8.0	-	-
MFHN	From 8.0	Halting debug	-
FLKK	From 8.0	M	-

B11.4.5 External debug request

R_{XZCP} When the PE is in Non-debug state, an external agent can signal an external debug request.

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.

R_{FGCV} The PE ignores external debug requests when it is in Debug state.

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.

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.

See also:

- *Debug event behavior* on page B11-241.
- DFSR.EXTERNAL.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XZCP	From 8.0	None	-
GTGX	From 8.0	M || Halting debug	-
FGCV	From 8.0	Halting debug	-
BXRD	From 8.0	-	-
WGMB	From 8.0	M	-

B11.5 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.

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 (R<n>), Special-purpose register, Floating-point Extension register, or DebugReturnAddress completes.

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.

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.

R_{VLVD} In Debug state, following a write to DCRDR 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.

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.

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.

R_{QLRN} The debugger can write to the EPSR.IT/ICI 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.

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.

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.

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.

RMDJX 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 (SCB).
- Indirect read of the Special-purpose register made for an access to any register through DCRDR or any register within the System Control Block.

IDMTG 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.

See also:
- *DCRDR, Debug Core Register Data Register* on page D1-922.
- *DCRSR, Debug Core Register Select Register* on page D1-923.

Rule	Architecture version	Required extensions	Notes
RMKS	From 8.0	Halting debug	-
QDCP	From 8.0	Halting debug	Floating-point registers are RES0 if FP is not implemented
VLVD	From 8.0	Halting debug	-
JKBB	From 8.0	Halting debug	-
QLRN	From 8.0	Halting debug	-
XRRQ	From 8.0	Halting debug	-
BMHD	From 8.0	Halting debug	-
MDJX	From 8.0	Halting debug	-

B11.6 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.

R$_{GGMJ}$ DebugReturnAddress is the address of the first instruction to be executed on exit from Debug state. This address indicates the point in the execution stream where the debug event was invoked. For a breakpoint this is the address identified by the breakpoint instruction. For all other debug events DebugReturnAddress is the address of the first instruction that both:

- In a simple sequential execution of the program, executes after the instruction that caused the debug event.
- Has not been executed, where the PE has executed all instructions that are earlier in a simple sequential execution of the program than the instruction indicated by DebugReturnAddress.

R$_{RTST}$ The Debugger can write to the DebugReturnAddress, and on exiting Debug state the PE starts executing from the updated address. The Debugger ensures that the EPSR.IT bits and the EPSR.ICI bits are consistent with the new DebugReturnAddress.

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.

R$_{HNKB}$ Exiting Debug state has no architecturally defined effect on the Event Register and exclusive monitors.

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.

R$_{FKXH}$ Before leaving Debug state caused by an imprecise entry into Debug state the system is reset.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BFGT	From 8.0	Halting debug	-
GGMJ	From 8.0	Halting debug	-
RTST	From 8.0	Halting debug	-
XCCB	From 8.0	Halting debug	-
HNKB	From 8.0	Halting debug	-
WKSD	From 8.0	Halting debug	-
FKXH	From 8.0	Halting debug	-

B11.7 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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QXLS	From 8.0	None	-

B11.7.1 Cross-halt event

R_{DLCV} When the PE enters Debug state, it signals to an external agent that it is entering Debug state.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DLCV	From 8.0	Halting debug	-

B11.7.2 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.

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.

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.

R_{NJQN} The PE ignores external restart requests when it is in Non-debug state.

See also:

- *Exiting Debug state* on page B11-256.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
ZKVW	From 8.0	Halting debug	-
WJST	From 8.0	Halting debug	-
NJQN	From 8.0	None	-

Chapter B12
Debug and Trace Components

This chapter specifies the Armv8-M debug and trace component rules. It contains the following sections:

B12.1 Instrumentation Trace Macrocell

B12.1.1 About the ITM

R_{GDNG}

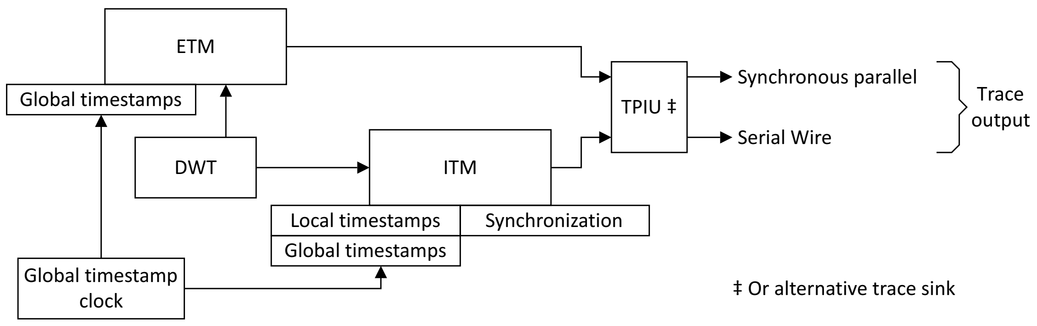
RGDNG The *Instrumentation Trace Macrocell* (ITM) provides a memory-mapped register interface that applications can use to generate Instrumentation packets.

IBXWJ The ITM is only available if the Main Extension is implemented.

RLMXS The ITM generates Instrumentation packets, Synchronization packets, and the following protocol packets:
- Overflow packets.
- Local timestamp packets.
- Global timestamp packets.
- Extension packets.

RXQRX 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.

IFQLR The following figure shows how the ITM relates to other debug components.



RBWJJ 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.

See also:
- *Global timestamping* on page B12-266.
- *Data Watchpoint and Trace unit* on page B12-270.
- Chapter F1 *ITM and DWT Packet Protocol Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GDNG	From 8.0	ITM	-
LMXS	From 8.0	ITM	-
XQRX	From 8.0	ITM	-
BWJJ	From 8.0	ITM	-

B12.1.2 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.

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.

R_{CGVD} If the ITM is disabled or not implemented, any Secure or Non-secure write to ITM_STIMn is ignored.

R_{NJTR} Unprivileged and privileged software can always read all ITM registers.

R_{FFXF} If the ITM is not implemented, the ITM registers are RAZ/WI.

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.

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].

R_{LNMW} Non-word-aligned accesses are UNPREDICTABLE.

R_{NQVK} ITM_TCR.ITMENA is a global enable bit for the ITM. A Cold reset clears this bit to 0, disabling the ITM.

R_{VRGP} The ITM_TERn registers provide an enable bit for each stimulus port.

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 packets from the output buffer to a trace sink.

R_{TCTH} If DEMCR.TRCENA == 0 or `NoninvasiveDebugAllowed()` == FALSE, the ITM does not generate trace.

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.

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.

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.

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.

R_{FXSL} Hardware source packets that are generated by the DWT unit 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 the DWT unit.

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.

R_{NFJN} Stalling does not affect the DWT counters.

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.

R_{CRKK} The ITM does not stall the PE in Secure state if `SecureHaltingDebugAllowed()` == FALSE.

R_{GRHW} The ITM does not stall the PE if `HaltingDebugAllowed()` == FALSE.

R_{BGCP} The ITM does not stall the PE in such a way as to deadlock the system.

R_{FRJG} The ITM does not stall the PE if the trace output is disabled.

R_{XRVL} The ITM does not stall for writes to the ITM_STIMn registers.

R_{HDLH} Instrumentation trace packets appear in the trace output in the order in which writes arrive at the ITM_STIMn registers.

R_{XNHX} It is IMPLEMENTATION DEFINED whether an ITM requires flushing of trace data to guarantee that data is output.

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.

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 ITM and DWT trace stream. To avoid trace stream corruption, before modifying the ITM_TCR.TraceBusID a debugger does the following:

- It clears the ITM_TCR.ITMENA bit to 0, to disable the ITM.
- It polls the ITM_TCR.BUSY bit until it returns to 0, indicating that the ITM is inactive.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
NKSC	From 8.0	ITM	-
MFDV	From 8.0	ITM	-
CGVD	From 8.0	ITM	-
NJTR	From 8.0	ITM	-
FFXF	From 8.0	ITM	-
CSFV	From 8.0	ITM	-
PTXV	From 8.0	ITM	-
LNMW	From 8.0	ITM	-
NQVK	From 8.0	ITM	-
VRGP	From 8.0	ITM	-
NTCR	From 8.0	ITM	-
TCTH	From 8.0	ITM	-
GRNM	From 8.0	ITM	-
SXNK	From 8.0	ITM	-
SRPP	From 8.0	ITM	-
XVVB	From 8.0	ITM	-
FXSL	From 8.0	ITM	-
RGCV	From 8.0	ITM	-
NFJN	From 8.0	ITM	-
TNDP	From 8.0	ITM	-
CRKK	From 8.0	ITM && S	-
GRHW	From 8.0	ITM	-
BGCP	From 8.0	ITM	-
FRJG	From 8.0	ITM	-
XRVL	From 8.0	ITM	-
HDLH	From 8.0	ITM	-
XNHX	From 8.0	ITM	-
TSXR	From 8.0	ITM	-
MKFS	From 8.0	ITM	-

B12.1.3 Timestamp support

R_{RVLT} Timestamps provide information on the timing of event generation regarding their visibility at a trace output port.

R_{TFDG} An Armv8-M PE can implement either or both of the following types of timestamp:

- Local timestamps.
- Global timestamps.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RVLT	From 8.0	ITM	-
TFDG	From 8.0	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.

R_{WGBG} The ITM generates the local timestamps from the timestamp counter in the ITM unit.

R_{XLBH} The timestamp counter size is an IMPLEMENTATION DEFINED value that is less than or equal to 28 bits.

R_{GPXT} It is IMPLEMENTATION DEFINED whether the ITM supports synchronous clocking of the timestamp counter mode.

R_{SRJH} It is IMPLEMENTATION DEFINED whether the ITM and TPIU support asynchronous clocking of the timestamp counter mode.

R_{GHPS} ITM_TCR.TSENA enables Local timestamp packet generation.

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.

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.

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.

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.

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.

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.

I_{SQLG} To prevent overflow, Arm recommends that the ITM emits a Local timestamp packet before the timestamp counter overflows.

See also:

- *Local timestamp clocking options*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RMXM	From 8.0	ITM	-
WGBG	From 8.0	ITM	-
XLBH	From 8.0	ITM	-
GPXT	From 8.0	ITM	-
SRJH	From 8.0	ITM	-
GHPS	From 8.0	ITM	-
FSWG	From 8.0	ITM	-
BRRL	From 8.0	ITM	-
XFRH	From 8.0	ITM	-
QJJB	From 8.0	ITM	-
NDCK	From 8.0	ITM	-
TBMX	From 8.0	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.

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.

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.

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.

R_{NGDW} When asynchronous clocking is implemented, whether the incoming clock signal can be divided before driving the local timestamping counter is IMPLEMENTATION DEFINED.

R_{RMTN} If the implementation supports division of the incoming asynchronous clock signal, ITM_TCR.TSPrescale sets the prescaler divide value.

R_{SKCP} Software only selects asynchronous clocking when the TPIU is configured to use an output mode that supports asynchronous clocking.

RJGCF 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.

See also:

- *Trace Port Interface Unit* on page B12-293.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	**Architecture version**	**Required extensions**	**Notes**
DSTG	From 8.0	ITM	-
BDWS	From 8.0	ITM	-
JDRD	From 8.0	ITM	-
NGDW	From 8.0	ITM	-
RMTN	From 8.0	ITM	-
SKCP	From 8.0	ITM	-
JGCF	From 8.0	ITM	-

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.

R_{HBWD} If an implementation includes Global timestamping, the ITM generates *Global timestamp* (GTS) packets, which are based on a global timestamp clock.

R_{KWQJ} The size of the global timestamp is either 48 bits or 64 bits. The choice between these two options is IMPLEMENTATION DEFINED.

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.

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.

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.

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.

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.

See also:

- *Synchronization support*.
- *Continuation bits* on page B12-268.
- Chapter F1 *ITM and DWT Packet Protocol Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HBWD	From 8.0	ITM	-
KWQJ	From 8.0	ITM	-
SRDF	From 8.0	ITM	-
VGBT	From 8.0	ITM	-
XQWL	From 8.0	ITM	-
DJLN	From 8.0	ITM	-
WDCX	From 8.0	ITM	-

B12.1.4 Synchronization support

I_{LRJT} An external debugger uses Synchronization packets to recover bit-to-byte alignment information in a serial data stream.

I_{LVGD} Synchronization packets are independent of timestamp packets.

I_{JNJV} Arm recommends that software disables Synchronization packets when using an asynchronous serial trace port, to reduce the data stream bandwidth.

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.

See also:

- DWT_CTRL.SYNCTAP.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RMND	From 8.0	ITM	-

B12.1.5 Continuation bits

I_{BFMX}

IBFMX 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.

RJNVH 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.

RXFTL 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.

RBBSF Each packet type defines its maximum packet length. Except for Global timestamp 2 and Synchronization packets, the longest defined packet is 5 bytes.

RDPJG The continuation bit, C, is defined as:

0	This is the last byte of the packet.
1	This is not the last byte of the packet.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
JNVH	From 8.0	ITM	-

Rule	Architecture version	Required extensions	Notes
XFTL	From 8.0	ITM	-
BBSF	From 8.0	ITM	-
DPJG	From 8.0	ITM	-

B12.2 Data Watchpoint and Trace unit

B12.2.1 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.

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.

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.

R_{CPXJ} If DWT_CTRL.NOTRCPKT is 1, there is no DWT trace support.

R_{FKFP} If DWT_CTRL.NOCYCCNT is 1, there is no cycle counter support.

R_{BKGF} If DWT_CTRL.NOPRFCNT is 1, there is no profiling counter support.

R_{HFTT} The DWT_CTRL.NUMCOMP field indicates the number of implemented DWT comparators, which is in the range 0-15.

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.

R_{SSWT} Data trace packets are only generated for comparators 0-3.

R_{CRHX} When a DWT implementation includes one or more comparators, which comparator features are supported, and by which comparators, is IMPLEMENTATION DEFINED.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
QQLQ	From 8.0	DWT-T && (M || Halting debug)	Some comparator matches require ETM
KBMX	From 8.0	DWT-T && M	-
DVJV	From 8.0	DWT-T && M && ITM	-
CPXJ	From 8.0	DWT-T	-
FKFP	From 8.0	DWT-T	-
BKGF	From 8.0	DWT-T	-
HFTT	From 8.0	DWT-T	-
WQLX	From 8.0	!M && DWT-T	-
SSWT	From 8.0	DWT-T	-
CRHX	From 8.0	DWT-T	-

B12.2.2 DWT unit operation

IWTSS 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.

RXBRD 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.

RDVCN A secure match can be generated by a cycle counter match in Secure state if DWT_CTRL.CYCDISS == 1.

RMGGT 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.

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.

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.

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.

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.

R$_{PKRK}$ If DEMCR.TRCENA == 0 or `NoninvasiveDebugAllowed()` == FALSE, DWT_CTRL.FOLDEVTENA, DWT_CTRL.LSUEVTENA, DWT_CTRL.SLEEPEVTENA, DWT_CTRL.EXCEVTENA, and DWT_CTRL.CPIEVTENA are ignored, and the PE behaves as if they have a value of 0.

R$_{GDMN}$ If DEMCR.TRCENA == 0 or `NoninvasiveDebugAllowed()` == FALSE, the DWT does not generate any trace packets.

R$_{FHWV}$ If `SecureNoninvasiveDebugAllowed()` == FALSE, DWT_CTRL.FOLDEVTENA, DWT_CTRL.LSUEVTENA, DWT_CTRL.SLEEPEVTENA, DWT_CTRL.EXCEVTENA, and DWT_CTRL.CPIEVTENA are ignored and the PE behaves as if they have a value of 0 in Secure state.

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.

R$_{DFWR}$ Exception trace packets appear in the same order as for a simple sequential execution of the exception handling.

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.

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.

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.

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.

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 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.

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.

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.

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.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
XBRD	From 8.0	DWT-T && S	-
DVCN	From 8.0	DWT-T && S	-
MGGT	From 8.0	DWT-T	-
GFLN	From 8.0	DWT-T &&S	-
HCFP	From 8.0	DWT-T && S	-
FFKV	From 8.0	DWT-T	-
VTNJ	From 8.0	DWT-T	-
PKRK	From 8.0	DWT-T	-
GDMN	From 8.0	DWT-T	-

Rule	Architecture version	Required extensions	Notes
FHWV	From 8.0	DWT-T && S	-
WSRR	From 8.0	DWT-T && S	-
DFWR	From 8.0	DWT-T	-
XDVS	From 8.0	DWT-T	-
RTJR	From 8.0	DWT-T && S	-
BRSR	From 8.0	DWT-T	-
NRGV	From 8.0	DWT-T	-
PJGW	From 8.0	DWT-T && M	-
FTBG	From 8.0	DWT-T && M && S	-
FNDW	From 8.0	DWT-T && (S || M && MPU)	-
PGJB	From 8.0	DWT-T	-
NQNR	From 8.0	DWT-T && S	-
SFSC	From 8.0	DWT-T	-

B12.2.3 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.

R_{GPLQ} Combinations of DWT_FUNCTION.MATCH and DWT_FUNCTION.ACTION that are not specified as valid combinations are reserved.

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 packet	Data trace data address packet	Data trace data value packet
Disabled	`0b0000`	`0bxx`	-	-	-	-	-
Cycle counter[a]	`0b0001`	`0b00`	-	-	-	-	-
		`0b01`	Yes	-	-	-	-
		`0b10`	-	Yes	-	-	-
		`0b11`	-	-	Yes	-	-
Instruction address	`0b0010`	`0b00`	-	-	-	-	-
		`0b01`	Yes	-	-	-	-
		`0b10`[a]	-	Yes	-	-	-
Data address	`0b01xx` (not `0b0111`)	`0b00`	-	-	-	-	-
		`0b01`	Yes	-	-	-	-
		`0b10`[a]	-	Yes	-	-	-
		`0b11`[a]	-	-	Yes	-	-
Data value[a]	`0b10xx` (not `0b1011`)	`0b00`	-	-	-	-	-
		`0b01`	Yes	-	-	-	-
		`0b10`	-	Yes	-	-	-
Data address with value[a]	`0b11xx` (not `0b1111`)	`0b10`	-	-	-	-	Yes
		`0b11`	-	-	Yes	-	Yes

a. Only if the Main Extension is implemented.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
MSPS	From 8.0	DWT-T	-
GPLQ	From 8.0	DWT-T	-
CNHN	From 8.0	DWT-T	-

Instruction address range

RDKHG 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*.

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		ACTION		Debug event	Data trace match packet	Data trace PC value packet	Data trace data address packet	Data trace data value packet
<n-1>	<n>	<n-1>	<n>					
`0b0000`	`0b0011`	`0bxx`	`0bxx`	-	-	-	-	-
`0b0010`	`0b0011`	`0b00`	`0b00`	-	-	-	-	-
		`0b00`	`0b11`[a]	-	-	Second	-	-
		`0b01`	`0b00`	First	-	-	-	-
		`0b10`[a]	`0b00`	-	First	-	-	-

a. Only if the Main Extension is implemented.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
DKHG	From 8.0	DWT-T	-
LNQD	From 8.0	DWT-T	-

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.*

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		Debug event	Data trace match packet	Data trace PC value packet	Data trace data address packet	Data trace data value packet
<n-1>	<n>	<n-1>	<n>					
`0b0000`	`0b0111`	`0bxx`	`0bxx`	-	-	-	-	-
`0b01xx` (not `0b0111`)	`0b0111`	`0b00`	`0b00`	-	-	-	-	-
		`0b00`	`0b11`[a]	-	-	-	Second	-
		`0b01`	`0b00`	First	-	-	-	-
		`0b10`[a]	`0b00`	-	First	-	-	-
		`0b11`[a]	`0b00`	-	-	First	-	-
		`0b11`[a]	`0b11`	-	-	First	Second	-
`0b11xx`[a] (not `0b1111`)	`0b0111`	`0b10`	`0b00`	-	-	-		First
		`0b10`	`0b11`	-	-	-	Second	First
		`0b11`	`0b00`	-	-	First		First
		`0b11`	`0b11`	-	-	First	Second	First

a. Only if the Main Extension is implemented.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
LDGR	From 8.0	DWT-T	-
PSBJ	From 8.0	DWT-T	-

Data value at specific address

RKFHV Matching data values at specific data addresses is possible only if the Main Extension is implemented.

RNNXD 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.*

RJKGJ The first comparator matches any access that matches the address. The second matches only accesses that match the address and the data value.

RNTSD 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		Debug event	Data trace match packet	Data trace PC value packet	Data trace data address packet	Data trace data value packet
<n-1>	<n>	<n-1>	<n>					
`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	-	-	-
		`0b10`	`0b10`	-	Both	-		-
		`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

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KFHV	From 8.0	DWT-T	-
NNXD	From 8.0	DWT-T	-
JKGJ	From 8.0	DWT-T	-
NTSD	From 8.0	DWT-T	-

B12.2.4 CMPMATCH trigger events

I_{VNCC} The **CMPMATCH** events signal watchpoint matches.

R_{PRJG} The implementation of **CMPMATCH** is IMPLEMENTATION DEFINED.

R_{FTWC} If an ETM is implemented, **CMPMATCH** events are output to the 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.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
PRJG	From 8.0	DWT-T	-
FTWC	From 8.0	DWT-T && ETM	-
TMZX	From 8.0	DWT-T	-
XXKM	From 8.0	DWT-T	-
GVHS	From 8.0	DWT-T	-

B12.2.5 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.

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.

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.

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.

R_{KLXM} For single instruction address matches, an instruction matches if the address of the first byte of the instruction matches the comparator address.

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.

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.

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.

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.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
GNVB	From 8.0	DWT-T	-
NQGR	From 8.0	DWT-T	-
KJJC	From 8.0	DWT-T	-
DSDT	From 8.0	DWT-T	-
KLXM	From 8.0	DWT-T	-
FXFM	From 8.0	DWT-T	-
DNKD	From 8.0	DWT-T	-
JNXZ	From 8.0	DWT-T	-
MLMQ	From 8.0	DWT-T	-
HMNX	From 8.0	DWT-T	-

Data address matching in detail

R$_{BPWC}$ For all Data Address matches, all bits of the comparator address are considered.

R$_{GSLX}$ Speculative reads might generate data address matches.

R$_{WWBH}$ Speculative writes do not generate data address matches.

R$_{VJFB}$ Prefetches into a cache do not generate data address matches.

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.

R$_{KHRF}$ For single data address matches, the comparator address is naturally aligned to DWT_FUNCTION.DATAVSIZE otherwise generation of watchpoint events is UNPREDICTABLE.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BPWC	From 8.0	DWT-T	-
GSLX	From 8.0	DWT-T	-
WWBH	From 8.0	DWT-T	-
VJFB	From 8.0	DWT-T	-
CMRP	From 8.0	DWT-T	-
KHRF	From 8.0	DWT-T	-
KKRJ	From 8.0	DWT-T	-
CFMR	From 8.0	DWT-T	-

Data value matching in detail

R_{BMSM} Data value matching is only possible if the Main Extension is implemented.

R_{FVFQ} Speculative reads might generate data value matches.

R_{VGJF} Speculative writes do not generate data value matches.

R_{MLFK} Prefetches into a cache do not generate data value matches.

R_{RMDB} For data value matches, if the access size is smaller than DWT_FUNCTION.DATAVSIZE, there is no match.

R_{ZDPM} For unlinked data value matches, an access matches if all bytes of any naturally-aligned subset of the access of the size that is specified by DWT_FUNCTION.DATAVSIZE match the data value in DWT_COMPn. The data value in DWT_COMPn is in little-endian order with respect to memory.

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.

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.

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.

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.

R_{KRCV} A Data value comparator that is linked to a Data address comparator does not change the behavior of the address comparator.

See also:

- `DWT_AddressCompare()`.
- `DWT_ValidMatch()`.
- `DWT_InstructionAddressMatch()`.
- `DWT_DataAddressMatch()`.
- `DWT_DataValueMatch()`.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BMSM	From 8.0	DWT-T	-
FVFQ	From 8.0	DWT-T	-
VGJF	From 8.0	DWT-T	-
MLFK	From 8.0	DWT-T	-
RMDB	From 8.0	DWT-T	-
ZDPM	From 8.0	DWT-T	-
SQKS	From 8.0	DWT-T	-
QRPW	From 8.0	DWT-T	-
QVRK	From 8.0	DWT-T	-
KRCV	From 8.0	DWT-T	-

B12.2.6 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.

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.

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.

R_{VJTK} The DWT does not match accesses from the DAP.

R_{MNBX} Any executed NOP or IT that matches an appropriately configured instruction address watchpoint causes a match.

R_{SLPX} It is IMPLEMENTATION DEFINED whether a failed `STREX` instruction can generate a data access match.

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.

R_{CSSQ} If an instruction or operation makes multiple or unaligned data accesses, then **CMPMATCH** is triggered for each matching access.

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.

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.

R$_{QCJL}$ The DWT can match on the address of an access that generates a BusFault.

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.

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.

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.

R$_{FRHP}$ DWT matches are generated asynchronously.

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.

R$_{HPGH}$ The effects of a DWT match never effect instructions appearing in program order before the operation that generates the match.

See also:

- *Tail-chaining* on page B3-93.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FRWG	From 8.0	DWT-T	-
DTHW	From 8.0	DWT-T	-
JQHW	From 8.0	DWT-T	-
VJTK	From 8.0	DWT-T	-
MNBX	From 8.0	DWT-T	-
SLPX	From 8.0	DWT-T	-
NHLN	From 8.0	DWT-T	-
CSSQ	From 8.0	DWT-T	-
VFXT	From 8.0	DWT-T	-
WJNR	From 8.0	DWT-T	-

Rule	Architecture version	Required extensions	Notes
QCJL	From 8.0	DWT-T	-
QVHL	From 8.0	DWT-T	-
KLFC	From 8.0	DWT-T	-
TQCF	From 8.0	DWT-T	-
FRHP	From 8.0	DWT-T	-
THHR	From 8.0	DWT-T	-
HPGH	From 8.0	DWT-T	-

B12.2.7 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

I_{MJXG} Arm recommends that the packet from the lowest-numbered comparator is output.

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.

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.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
HDKK	From 8.0	DWT-T	-
WSKK	From 8.0	DWT-T	-
XCNB	From 8.0	DWT-T	-
XVBT	From 8.0	DWT-T	-
QSCF	From 8.0	DWT-T	-
KXBL	From 8.0	DWT-T	-
QWQS	From 8.0	DWT-T	-
CPXW	From 8.0	DWT-T	-
QXBC	From 8.0	DWT-T	-
RHNF	From 8.0	DWT-T	-
DKMV	From 8.0	DWT-T	-
LNBW	From 8.0	DWT-T	-

B12.2.8 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.

R_{KRFP} When implemented and enabled, CYCCNT increments on each cycle of the PE clock.

R_{NFJW} When the counter overflows it transparently wraps to zero.

R_{GXJK} DWT_CTRL.CYCCNTENA enables the CYCCNT counter.

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.

I_{MGGL} Periodic PC sample packets are not the same as the Data trace PC value packets that are generated by the DWT comparators.

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.

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

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.

R_{XFRM} When either DWT_CTRL.CYCEVTENA or DWT_CTRL.PCSAMPLENA 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.

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.

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.

R_{KNHF} Disabling CYCCNT stops POSTCNT.

R_{TMHN} Writes to DWT_CTRL.POSTINIT are ignored if either DWT_CTRL.PCSAMPLENA was set to 1 or DWT_CTRL.CYCEVTENA was set to 1 prior to the write.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
SVPW	From 8.0	DWT-T	-
KRFP	From 8.0	DWT-T	-
NFJW	From 8.0	DWT-T	-
GXJK	From 8.0	DWT-T	-
BKCG	From 8.0	DWT-T	-
DKGR	From 8.0	DWT-T	-
RNTV	From 8.0	DWT-T	-
SXKK	From 8.0	DWT-T	-
XFRM	From 8.0	DWT-T	-
KNHF	From 8.0	DWT-T	-
TMHN	From 8.0	DWT-T	-

B12.2.9 Profiling counter support

I_{HXPV} If the Main Extension is implemented profiling counter support is an optional Non-invasive debug feature.

R_{WHWR} If profiling counter support is implemented the DWT provides five 8-bit Event counters for software profiling:

- DWT_FOLDCNT.
- DWT_LSUCNT.
- DWT_EXCCNT.
- DWT_SLEEPCNT.
- DWT_CPICNT.

R_{GLMJ} Event counters do not increment when the PE is halted.

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.

R_{WMNV} The Event counters use the same definition of cycle in particular when counting cycles in power-saving modes.

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.

I_{LMDG} Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the Event counters are expected.

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.

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.

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.

R_{LHMB} Setting one of the enable bits to 1 clears the corresponding counter to 0.

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_LSUCNT.
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.

See also:

- *Trace Port Interface Unit* on page B12-293.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
WHWR	From 8.0	DWT-T,	-
GLMJ	From 8.0	DWT-T	-
BRGW	From 8.0	DWT-T	-
WMNV	From 8.0	DWT-T	-
LMDG	From 8.0	DWT-T	-
LMPK	From 8.0	DWT-T	-
LHMB	From 8.0	DWT-T	-

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.

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.

R_{VPXK} If the DWT unit attempts to generate a packet when its output buffer is full, an Overflow packet is output.

R_{SFFL} The size of the DWT output buffer is IMPLEMENTATION DEFINED.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
KWDH	From 8.0	DWT-T	-
HKTL	From 8.0	DWT-T	-
VPXK	From 8.0	DWT-T	-
SFFL	From 8.0	DWT-T	-

B12.2.10 Program Counter sampling support

R_{FXWL} Program Counter sampling is an optional component provided through DWT_PCSR.

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.

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.

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.

I_{KCBH} Arm recommends that instructions that fail the condition code check are considered as committed instructions.

R_{WPMF} DWT_PCSR is able to sample references to branch targets. It is IMPLEMENTATION DEFINED whether it can sample references to other instructions.

I_{SJVK} Arm recommends that DWT_PCSR can sample a reference to any instruction.

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.

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

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.

I$_{KVJM}$ Arm strongly recommends that an implementation document any particular scenarios where significant inaccuracies in the sampled data are expected.

R$_{JMVS}$ When DEMCR.TRCENA is set to 0 any read of DWT_PCSR returns an UNKNOWN value.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FXWL	From 8.0	DWT-T	-
NGNT	From 8.0	DWT-T	-
WPMF	From 8.0	DWT-T	-
LMDG	From 8.0	DWT-T	-
NWKP	From 8.0	DWT-T	-
KVJM	From 8.0	DWT-T	-
JMVS	From 8.0	DWT-T	-

B12.3 Embedded Trace Macrocell

I_{LCCX} An *Embedded Trace Macrocell* (ETM) is an optional non-invasive debug feature of an Armv8-M implementation.

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 or later
	ETMv4, version 4.2 or later	ETMv4, version 4.0 or later

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.

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.

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.

R_{WPBN} If the ETMv3 is implemented the debugger programs the ETMTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

R_{TJSF} If the ETMv4 is implemented the debugger programs the TRCTRACEIDR with a unique nonzero Trace ID for the ETM trace stream.

R_{WSTB} The ETM is not directly affected by DEMCR.TRCENA being set to 0.

See also:
- *ARM® CoreSight™ Architecture Specification.*
- *CMPMATCH trigger events* on page B12-279.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
NGTT	From 8.0	ETM	-
LPJM	From 8.0	ETM	-
NLNS	From 8.0	ETM	-
NJDK	From 8.0	ETM	-

Rule	Architecture version	Required extensions	Notes
WPBN	From 8.0	ETM	-
TJSF	From 8.0	ETM	-
WSTB	From 8.0	ETM	-

B12.4 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.

R$_{CRTQ}$ It is IMPLEMENTATION DEFINED whether the TPIU supports a parallel trace port output.

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.

R$_{LKQT}$ It is IMPLEMENTATION DEFINED whether the TPIU supports medium-speed asynchronous serial port output using Manchester encoding.

I$_{SDDK}$ Arm recommends that the TPIU provides both parallel and asynchronous serial ports, for maximum flexibility with external capture devices.

R$_{HJXK}$ Whether the trace port clock is synchronous to the PE clock is IMPLEMENTATION DEFINED.

R$_{PKKS}$ It is IMPLEMENTATION DEFINED whether the TPIU is reset by a Cold reset or has an independent Cold reset.

R$_{JBKJ}$ Software ensures that all trace is output and flushed to the trace sink before setting the DEMCR.TRCENA bit to 0.

R$_{STLV}$ The TPIU is not directly affected by DEMCR.TRCENA being set to 0 or `NoninvasiveDebugAllowed()` being FALSE.

R$_{JLCQ}$ The output formatting modes that are supported by the TPIU are IMPLEMENTATION DEFINED. They are:

- Bypass.
- Continuous.

R$_{DMFP}$ Bypass mode is only supported if a serial port output is supported.

R$_{RRJP}$ Continuous mode is supported if the parallel trace port is implemented. Continuous mode is selected when the parallel trace port is used.

R$_{FCFT}$ Continuous mode is supported if the ETM is implemented. Continuous mode is selected when the ETM is used.

See also:

- *TPIU_FFCR, TPIU Formatter and Flush Control Register* on page D1-1155.
- *Instrumentation Trace Macrocell* on page B12-260.
- *Embedded Trace Macrocell* on page B12-291.
- Chapter B1 *Resets*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
CRTQ	From 8.0	TPIU	-
GTRP	From 8.0	TPIU	-
LKQT	From 8.0	TPIU	-
HJXK	From 8.0	TPIU	-
PKKS	From 8.0	TPIU	-
JBKJ	From 8.0	TPIU	-

Rule	Architecture version	Required extensions	Notes
STLV	From 8.0	TPIU	-
JLCQ	From 8.0	TPIU	-
DMFP	From 8.0	TPIU	-
RRJP	From 8.0	TPIU	-
FCFT	From 8.0	TPIU	-

B12.5 Flash Patch and Breakpoint unit

B12.5.1 About the FPB unit

R_{FTWL} The *Flash Patch and Breakpoint* (FPB) unit supports setting breakpoints on instruction fetches.

I_{BPFS} The name Flash Patch and Breakpoint unit is historical and the architecture does not support remapping functionality.

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.

See also:
- Chapter B6 *The System Address Map*.
- *DWT trace restrictions and relaxations* on page B12-284.
- Chapter D1 *Register Specification*.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
FTWL	From 8.0	FPB	-
GDWW	From 8.0	FPB	-

B12.5.2 FPB unit operation

R_{RKFD} The FPB contains the following register types:
- A general control register, FP_CTRL.
- Comparator registers.

R_{BKKW} Each implemented instruction address comparator supports breakpoint generation.

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.

R_{CKBL} When configured for breakpoint generation, instruction address comparators can be configured to match any halfword-aligned addresses in the whole address map.

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 access from the DAP.

I_{CNBW} Bit[0] of each instruction fetch address is always 0.

RCJKK 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.

RWSXN 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.

RXKJW 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.

RHXMP 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.

RBFPK 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.

RCLNV 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.

See also:

- *Halting debug authentication* on page B11-234.
- *About debug events* on page B11-241.
- `BKPTInstrDebugEvent()`.
- `FPB_BreakpointMatch()`.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
RKFD	From 8.0	FPB	-
BKKW	From 8.0	FPB	-
FNQF	From 8.0	FPB	-

Rule	Architecture version	Required extensions	Notes
CKBL	From 8.0	FPB	-
XPXS	From 8.0	FPB	-
CJKK	From 8.0	FPB	-
WSXN	From 8.0	FPB	-
XKJW	From 8.0	FPB	-
HXMP	From 8.0	FPB	-
BFPK	From 8.0	FPB	-
CLNV	From 8.0	FPB	-

Cache maintenance

R_{BWSW} Instruction caches are not permitted to cache breakpoints that are generated by a Flash Patch and Breakpoint unit.

The following table lists the individual rules in this section and the extensions that must be implemented in order for the rule to apply.

Rule	Architecture version	Required extensions	Notes
BWSW	From 8.0	FPB	-

Part C
Armv8-M Instruction Set

Chapter C1
Instruction Set Overview

This chapter provides a definition of the *instruction descriptions* contained in Chapter C2 *Instruction Specification*. It contains the following sections:

C1.1 Instruction set

There is one instruction set, called T32.

For more information about the T32 instruction set encoding, see *Top level T32 instruction set encoding* on page C2-324.

See also:

- *Instruction set encoding information* on page C1-315.
- Chapter C2 *Instruction Specification*. See this for descriptions of each T32 instruction.

C1.2 Format of instruction descriptions

Each instruction description in Chapter C2 has the following content:

1. A title.
2. A short description.
3. The instruction encoding or encodings.
4. Any alias conditions, if applicable.
5. A list of the assembler symbols for the instruction.
6. Pseudocode describing how the instruction operates.
7. Notes, if applicable.

C1.2.1 The title

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`.

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 there are the following titles for different forms of the `ADD` instruction:

- *ADD (SP plus immediate)* on page C2-378.
- *ADD (SP plus register)* on page C2-380.
- *ADD (immediate)* on page C2-383.
- *ADD (immediate, to PC)* on page C2-386.
- *ADD (register)* on page C2-388.

Where an instruction has more than one variant, the descriptions might be combined, for example for `CDP` and `CDP2`.

C1.2.2 A short description

This briefly describes the function of the instruction. The description is not a complete description of the instruction and must be read in conjunction with the rest of the instruction description.

C1.2.3 The instruction encoding or encodings

This 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.

The instruction alignment and byte ordering figure in *Endianness* on page B5-135 shows the alignment for both 16-bit encodings and 32-bit encodings. The left-hand halfword in the diagram is called hw1 for a 16-bit encoding. The left-hand halfword in the diagram is called hw1 and the right-hand halfword is called hw2 for a 32-bit encoding.

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.*"

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.

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.

Each encoding diagram, and its associated assembler syntax prototypes, is followed by encoding-specific pseudocode that translates the fields of that encoding into inputs for the encoding-independent pseudocode that describes the operation of the instruction. See *Pseudocode describing how the instruction operates* on page C1-304.

C1.2.4 Any alias conditions, if applicable

This is 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.

Arm recommends that if a disassembler outputs the alias syntax, it consistently outputs the alias syntax.

C1.2.5 A list of the assembler symbols for the instruction

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.

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.

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.

Some assembler syntax prototype fields are standardized across all or most instructions.

See also:

- *Standard assembler syntax fields* on page C1-310.

C1.2.6 Pseudocode describing how the instruction operates

The *Operation* subsection of the instruction description contains this pseudocode.

It is encoding-independent pseudocode that provides a precise description of what the instruction does.

Note

For a description of Arm pseudocode, see Chapter E1 *Arm Pseudocode Definition*.

Where the pseudocode describes UNPREDICTABLE behavior the constraints on that behavior are described in the Operation section.

C1.2.7 Exceptions

The Exceptions subsection contains a list of the exceptional conditions that can be caused by execution of the instruction. For a list of the possible exceptions, see:

- *Exception numbers and exception priority numbers* on page B3-53.
- *The IEEE 754 floating-point exceptions* on page B4-123.

C1.2.8 Notes

Where appropriate, additional notes about the instruction appear under further subheadings.

C1.3 Pseudocode for instruction descriptions

Each instruction description includes pseudocode that provides a precise description of what the instruction does, subject to the limitations described in *General limitations of Arm pseudocode* on page E1-1186 and *Limitations of the instruction pseudocode*.

In the instruction pseudocode, instruction fields are referred to by the names shown in the encoding diagram for the instruction. *Instruction encoding diagrams and instruction pseudocode* gives more information about the pseudocode provided for each instruction.

C1.3.1 Instruction encoding diagrams and instruction pseudocode

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, either at its start or only after a condition code check performed by `if ConditionPassed() then`.

An encoding diagram specifies each bit of the instruction as one of the following:

- A mandatory 0 or 1, represented in the diagram as 0 or 1. If this bit does not have this value, the encoding corresponds to a different instruction.
- A *should be* 0 or *should be* 1, represented in the diagram as (0) or (1). If this bit does not have this value, the instruction is CONSTRAINED UNPREDICTABLE. For more information, see the *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.
- A named single bit or a bit in a named multi-bit field.

An encoding diagram matches an instruction if all mandatory bits are identical in the encoding diagram and the instruction,

C1.3.2 Pseudocode descriptions of operations on general-purpose registers and the PC

In pseudocode, the uses of the `R[]` function are:

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

R on page E2-1318 shows the function prototypes.

C1.3.3 Limitations of the instruction pseudocode

The pseudocode descriptions of instruction functionality have a number of limitations.

These limitations include:

- Pseudocode does not describe the ordering requirements when an instruction generates multiple memory accesses. For a description of the ordering requirements on memory accesses see *Ordering requirements for memory accesses* on page B5-147.
- Pseudocode does not describe 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. *Conditional execution of undefined instructions* on page C1-312 describes the exact rules.

- 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. *Priority of floating-point exceptions relative to other floating-point exceptions* on page B4-129 describes the exact rules.
- 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. To determine that, see the descriptions of the exceptions in *Exception handling* on page B3-76 and *Exception return* on page B3-87.

C1.4 Unified Assembler Language

This manual uses the Arm *Unified Assembler Language* (UAL). This assembly language syntax provides a canonical form for all T32 instructions.

UAL describes the syntax for the mnemonic and the operands of each instruction. For example:

```
BX {<c>}{<q>}<Rm>
```

In this example, `BX` is the mnemonic, and `{<c>}{<q>}<Rm>` are the operands.

Note

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.

UAL includes *instruction selection* rules that specify which instruction encoding is selected when more than one can provide the required functionality, see the definition of `<c>` in *Standard assembler syntax fields* on page C1-310.

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

C1.4.1 Conditional instructions

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.

For example:

```
ITTEE EQ
ADDEQ R0, R1
SUBEQ R2, R3
ADDNE R4, R5
SUBNE R6, R7
```

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.

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.

For more information, see *IT* on page C2-441.

C1.4.2 Use of labels in UAL instruction syntax

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

1. Calculate the `PC` or `Align(PC,4)` value of the instruction. The `PC` value of an instruction is its address plus 4 for a 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.

Note

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

The 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 0 from the `Align(PC,4)` value cannot be specified by the normal syntax.

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

Note

Arm recommends that where possible, the alias is used.

C1.4.3 Using syntax information

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.

For each instruction encoding that belongs to a target instruction set, an assembler can use the information in the encoding to determine whether it can use that particular encoding to encode the instruction requested by the UAL source. If multiple encodings can encode the instruction, then:

- If both a 16-bit encoding and a 32-bit encoding can encode the instruction, the architecturally preferred encoding is the 16-bit encoding. This means that the assembler must use the 16-bit encoding instead of the 32-bit encoding.
- If multiple encodings of the same width can encode the instruction, the assembler syntax indicates the preferred encoding, and how software can select other encodings if required. Each encoding also documents UAL syntax that selects it in preference to any other encoding. If no encodings of the target instruction set can encode the instruction requested by the UAL source, the assembler normally generates an error that indicates that the instruction is not available in that instruction set.

C1.5 Standard assembler syntax fields

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`). For details see *Conditional instructions* on page C1-308.
`<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.

C1.6 Conditional execution

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

Most 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* on page C2-404 for details.
- A 32-bit conditional branch instruction, with a branch range of approximately ± 1MB. See *B* on page C2-404 for details.
- 16-bit Compare and Branch on Zero and Compare and Branch on Nonzero instructions, with a branch range of +4 to +130 bytes. See *CBNZ, CBZ* on page C2-416 for details.
- A 16-bit If-Then instruction that makes up to four following instructions conditional. See *IT* on page C2-441 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.

In T32 instructions, the condition (if it is not `AL`) is encoded in a preceding `IT` instruction, other than `B`, `CBZ`, and `CBNZ`. Some conditional branch instructions do not require a preceding `IT` instruction, and include a condition code in their encoding.

Table C1-1 shows the conditions that are available for conditionally executed instructions.

Table C1-1 Condition codes

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	APSR condition flags
`0000`	`EQ`	Equal	Equal	Z == 1
`0001`	`NE`	Not equal	Not equal, or unordered	Z == 0
`0010`	`CS` [b]	Carry set	Greater than, equal, or unordered	C == 1
`0011`	`CC` [c]	Carry clear	Less than	C == 0
`0100`	`MI`	Minus, negative	Less than	N == 1
`0101`	`PL`	Plus, positive or zero	Greater than, equal, or unordered	N == 0
`0110`	`VS`	Overflow	Unordered	V == 1
`0111`	`VC`	No overflow	Not unordered	V == 0
`1000`	`HI`	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
`1001`	`LS`	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
`1010`	`GE`	Signed greater than or equal	Greater than or equal	N == V
`1011`	`LT`	Signed less than	Less than, or unordered	N != V
`1100`	`GT`	Signed greater than	Greater than	Z == 0 and N == V
`1101`	`LE`	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
`1110`	None (`AL`) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. `HS` (unsigned higher or same) is a synonym for `CS`.
c. `LO` (unsigned lower) is a synonym for `CC`.
d. `AL` is an optional mnemonic extension for always, except in `IT` instructions. See *IT* on page C2-441 for details.

C1.6.1 Pseudocode details of conditional execution

The `CurrentCond()` pseudocode function prototype returns a 4-bit condition specifier as follows:

- For the T1 and T3 encodings of the Branch instruction shown in *B* on page C2-404, 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.

  For more information, see *ITSTATE*.

The `ConditionPassed()` function uses this condition specifier and the APSR condition flags to determine whether the instruction must be executed.

C1.6.2 Conditional execution of undefined instructions

The conditional execution described in *Conditional execution* on page C1-311 applies to all instructions. This includes undefined instructions and other instructions that would cause entry to the UsageFault on the Undefined Instruction exception.

If such an instruction fails its condition code check, the behavior depends on the potential cause of entry to the UsageFault, as follows:

- If the potential cause is the execution of the instruction itself and depends on data values used by the instruction, the instruction executes a `NOP` and does not cause an UsageFault.
- In the following cases, it is IMPLEMENTATION DEFINED whether the instructions executes as a `NOP` or causes an Undefined Instruction exception:
  - The potential cause is the execution of an earlier System register instruction or floating-point instruction.
  - The potential cause is the execution of the instruction itself without dependence on the data values used by the instruction.

  An implementation must handle all such cases in the same way.

C1.6.3 Interaction of undefined instruction behavior with UNPREDICTABLE or CONSTRAINED UNPREDICTABLE instruction behavior

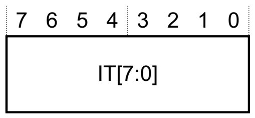
If this manual describes an instruction as both:

- UNPREDICTABLE and UNDEFINED, then the instruction is UNPREDICTABLE.
- CONSTRAINED UNPREDICTABLE and UNDEFINED, then the instruction is CONSTRAINED UNPREDICTABLE.

C1.6.4 ITSTATE

ITSTATE is held in EPSR, see *Execution Program Status Register (EPSR)* on page B3-46.

The bit assignments of the ITSTATE register are:



This register holds the If-Then Execution state bits for the T32 `IT` instruction. See *IT* on page C2-441 for a description of the `IT` instruction and the associated IT block.

ITSTATE divides into two subfields:

IT[7:5] Holds the *base condition* for the current IT block. The base condition is the top 3 bits of the condition specified by the `IT` instruction.

This subfield is `0b000` when no IT block is active.

IT[4:0] Encodes:

- The size of the IT block. This is the number of instructions that are to be conditionally executed. The size of the block is 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.

Note

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.

When an IT instruction is executed, IT bits[7:0] are set according to the condition in the instruction, and the *Then* and *Else* (`T` and `E`) parameters in the instruction. See *IT* on page C2-441 for more information.

An instruction in an IT block is conditional. See *Conditional instructions* on page C1-308. 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.

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
```

Note

Instructions that can complete their normal execution by branching are only permitted in an IT block as its last instruction, and so always result in `ITSTATE` advancing to normal execution.

In Table C1-2, *P* represents the base condition or the inverse of the base condition.

Table C1-2 Effect of IT Execution state bits

	IT bits [a]					
[7:5]	[4]	[3]	[2]	[1]	[0]	
cond_base	P1	P2	P3	P4	1	Entry point for 4-instruction IT block
cond_base	P1	P2	P3	1	0	Entry point for 3-instruction IT block
cond_base	P1	P2	1	0	0	Entry point for 2-instruction IT block
cond_base	P1	1	0	0	0	Entry point for 1-instruction IT block
`000`	0	0	0	0	0	Normal execution, not in an IT block

a. Combinations of the IT bits not shown in this table are reserved.

Pseudocode details of ITSTATE operation

`ITAdvance()` describes how ITSTATE advances after normal execution.

`InITBlock` and `LastInITBlock` test whether the current instruction is in an IT block, and whether it is the last instruction of an IT block.

C1.6.5 Branching into and out of an IT block

Execution of an instruction outside of an IT block with ITSTATE set to a non-zero IT value is UNPREDICTABLE.

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.

C1.7 Instruction set encoding information

C1.7.1 UNDEFINED and UNPREDICTABLE instruction set space

An attempt to execute an unallocated instruction results in either:

- UNPREDICTABLE behavior. The instruction is described as UNPREDICTABLE.
- An UNDEFINSTR Usage Fault. The instruction is described as UNDEFINED.

An instruction is UNDEFINED if it is declared as UNDEFINED in an instruction description in Chapter C2.

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 in Chapter C2.

Unless otherwise specified, a T32 instruction that is provided by one or more of the architecture extensions is either UNPREDICTABLE or UNDEFINED in an implementation that does not include those extensions. See the individual instruction descriptions for details.

C1.7.2 Use of 0b1111 as a register specifier

The use of `0b1111` as a register specifier is not normally permitted in T32 instructions. When a value of `0b1111` is permitted, 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).

For register writes, these meanings are:

- The PC can be specified as the destination register of an `LDR` instruction. This is done by encoding Rt as `0b1111`. The loaded value is treated as an address, and the effect of execution is a branch to that address. bit [0] of the loaded value selects the Execution state after the branch and must have the value 1.
- Discard the result of a calculation. This is done in some cases when one instruction is a special case of another, more general instruction, but with the result discarded. In these cases, the instructions are listed on separate pages, with a special case in the pseudocode for the more general instruction cross-referencing the other page. An example of this is the descriptions of TST (register) and AND (register).
- If the destination register specifier of an `LDRB`, `LDRH`, `LDRSB`, or `LDRSH` instruction is `0b1111`, the instruction is a memory hint instead of a load operation.
- If the destination register specifier of an `MRC` instruction is `0b1111`, bits [31:28] of the value transferred from the coprocessor are written to the N, Z, C, and V flags in the APSR, and bits [27:0] are discarded.

C1.7.3 Use of 0b1101 as a register specifier

R13 is defined in the T32 instruction set so that its use is primarily as a stack pointer, and R13 is normally identified as SP in T32 instructions.

The following subsections describe the restrictions that apply to using SP:

- *SP[1:0] definition*.
- *32-bit T32 instruction support for SP*.
- *16-bit T32 instruction support for SP*.

SP[1:0] definition

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.

32-bit T32 instruction support for SP

32-bit T32 instruction support for SP 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:

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

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

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

- 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.

Use of the SP as a general-purpose register in any other case is UNPREDICTABLE.

16-bit T32 instruction support for SP

For 16-bit data processing instructions that affect general-purpose registers R8-R15, SP can only be used as described in *32-bit T32 instruction support for SP*. Arm deprecates any other use. This affects the high register forms of `CMP` and `ADD`, where Arm deprecates the use of SP as Rm.

C1.7.4 Branching

Writing an address to the PC causes either a simple branch to that address or an *interworking* branch.

A simple branch is performed by BranchWritePC.

An interworking branch is performed by BXWritePC.

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`.

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`.

Table C1-3 summarizes the branch instructions in the T32 instruction set. In addition to providing for changes in the flow of execution, some branch instructions can change the Security state.

Table C1-3 Branch instructions

Instruction	See	Range, T32
Branch to target address	*B* on page C2-404	±16MB
Compare and Branch on Nonzero, Compare and Branch on Zero	*CBNZ, CBZ* on page C2-416	0-126 bytes
Call a subroutine	*BL* on page C2-412	±16MB
Call a subroutine, optionally change Security state	*BLX, BLXNS* on page C2-413	Any
Branch to target address, change to Non-secure state	*BX, BXNS* on page C2-415	Any
Table Branch (byte offsets) Table Branch (halfword offsets)	*TBB, TBH* on page C2-713	0-510 bytes 0-131070 bytes

Branches to loaded and calculated addresses can be performed by `LDR`, `LDM` and data-processing instructions. For details, see Chapter C2 *Instruction Specification*.

In addition to the branch instructions shown in Table C1-3, a load instruction that targets the PC behaves as a branch instruction.

C1.8 Modified immediate constants

The encoding of modified immediate constants in T32 instructions is:

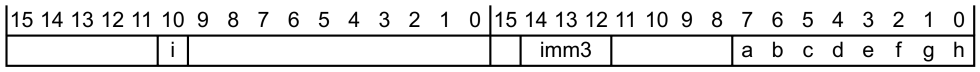


Table C1-4 shows the range of modified immediate constants available in T32 data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

Table C1-4 Encoding of modified immediates in T32 data-processing instructions

i:imm3:a	<const> [a]	Carry flag set[b]
`0000x`	`00000000 00000000 00000000 abcdefgh`	No
`0001x`	`00000000 abcdefgh 00000000 abcdefgh` [c]	No
`0010x`	`abcdefgh 00000000 abcdefgh 00000000` [c]	No
`0011x`	`abcdefgh abcdefgh abcdefgh abcdefgh` [c]	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

a. 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.

b. Applies only if a logical operation with a modified immediate constant can set the flags.

c. UNPREDICTABLE if abcdefgh == 00000000.

C1.8.1 Operation of modified immediate constants

T32ExpandImm on page E2-1336 and *T32ExpandImm_C* on page E2-1337 show the operation of modified immediate constants.

The description of immediate constants in data processing include constant values that were UNPREDICTABLE in Armv7. *Top level T32 instruction set encoding* on page C2-324 describes instructions as {`hw1`, `hw2`}, where `hw1` is the left-hand halfword in the 32-bit encoding diagram for the instruction. The UNPREDICTABLE cases are where both:

- `hw2` [7:0] == `0b0000000`.
- `hw1` [10] == `0` and either:
  - `hw2` [14:12] == `0b001`.
  - `hw2` [14:12] == `0b010`.
  - `hw2` [14:12] == `0b011`.

C1.9 NOP-compatible hint instructions

A hint instruction only provides an indication to the PE. It is not required that the PE perform an operation on a hint instruction.

A NOP-compatible hint instruction either:

- Acts as a `NOP` (No Operation) instruction.
- Is reserved.

A PE without the Main Extension only supports the 16-bit encodings of the Armv8-M NOP-compatible hint instructions. A PE with the Main Extension supports both the 16-bit and the 32-bit encodings of the Armv8-M NOP-compatible hint instructions.

- For information on the 16-bit encodings see *Hints* on page C2-333.
- For information on the 32-bit encodings see *Hints* on page C2-348.

C1.10 Instruction set, interworking support

In the A profile version of the ARMv8 architecture, ARMv8-A, the *AArch32 Execution state* supports two instruction sets, T32 and A32, and software can use *interworking branches* to select which of these to execute.

In Armv8-M, the following instructions are interworking branches:

- `BX` and `BLX`.
- `POP (multiple registers)` and all forms of `LDM`, when the register list includes the PC.
- `LDR (immediate)`, `LDR (literal)`, and `LDR (register)`, with <Rt> equal to the PC.

This means that the value of bit[0] for these instructions is not stored in the PC. Instead, it selects the instruction set that is executed after the branch.

In Armv8-M, if bit [0] of an interworking address is:

0 EPSR.T is assigned the value `0b0`, causing the PE to take an INVSTATE UsageFault on the next instruction it attempts to execute.

1 EPSR.T is assigned the value `0b1`. The instruction set state is T32 state and all instructions are decoded as T32 instructions.

Bit[0] of the PC is always 0.

See also:

- *Instruction set* on page C1-302.
- *BXWritePC* on page E2-1215.

C1.11 Instruction set, interstating support

In Armv8-M, the following instructions are *interstating branches*:

- `BXNS` and `BLXNS`.

This means that the value of bit[0] for these instructions is not stored in the PC. Instead, it selects the target Security state.

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.

Bit[0] of the PC is always 0.

Interstating branches are UNDEFINED when executing in Non-secure state.

See also:

- *Security state transitions* on page B3-71.

C1.12 SBZ or SBO fields in instructions

Many of the instructions have (0) or (1) in the instruction decode to indicate *Should-Be-Zero*, SBZ, or *Should-Be-One*, SBO. 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* on page C2-455.
- *LDMDB, LDMEA* on page C2-459.
- *LDR (immediate)* on page C2-462.
- *LDRB (literal)* on page C2-473.
- *LDRD (immediate)* on page C2-477.
- *LDRH (literal)* on page C2-487.
- *LDRSB (literal)* on page C2-493.
- *LDRSH (literal)* on page C2-500.
- *POP (multiple registers)* on page C2-571.
- *PUSH (multiple registers)* on page C2-574.
- *SDIV* on page C2-613.
- *STM, STMIA, STMEA* on page C2-662.
- *STMDB, STMFD* on page C2-665.
- *UDIV* on page C2-727.

Chapter C2
Instruction Specification

This chapter specifies the Armv8-M instruction set. It contains the following sections:

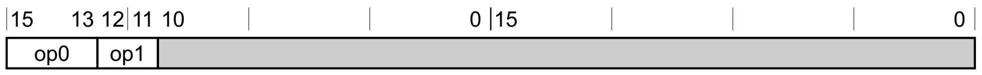
C2.1 Top level T32 instruction set encoding

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.



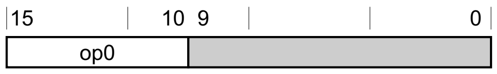
Decode fields		Decode group or instruction page
op0	op1	
`!= 111`	`-`	*16-bit T32 instruction encoding* on page C2-325
`111`	`00`	B - T2 variant
`111`	`!= 00`	*32-bit T32 instruction encoding* on page C2-336

C2.2 16-bit T32 instruction encoding

This section describes the encoding of the 16-bit T32 instruction encoding group. This section is decoded from *Top level T32 instruction set encoding* on page C2-324.

Note

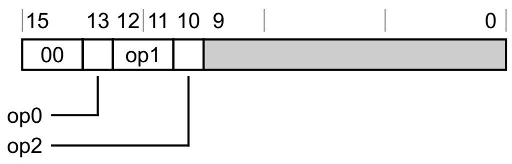
In the decode tables in this section, an entry of - for a field value means the value of the field does not affect the decoding.



Decode fields op0	Decode group or instruction page
00xxxx	*Shift (immediate), add, subtract, move, and compare*
010000	*Data-processing (two low registers)* on page C2-327
010001	*Special data instructions and branch and exchange* on page C2-328
01001x	LDR (literal) - T1 variant
0101xx	*Load/store (register offset)* on page C2-329
011xxx	*Load/store word/byte (immediate offset)* on page C2-329
1000xx	*Load/store halfword (immediate offset)* on page C2-330
1001xx	*Load/store (SP-relative)* on page C2-330
1010xx	*Add PC/SP (immediate)* on page C2-331
1011xx	*Miscellaneous 16-bit instructions* on page C2-331
1100xx	*Load/store multiple* on page C2-334
1101xx	*Conditional branch, and Supervisor Call* on page C2-334

C2.2.1 Shift (immediate), add, subtract, move, and compare

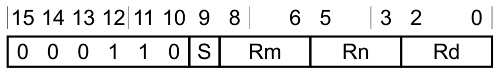
This section describes the encoding of the Shift (immediate), add, subtract, move, and compare group. The encodings in this section are decoded from *16-bit T32 instruction encoding*.



Decode fields			Decode group or instruction page
op0	op1	op2	
0	11	0	*Add, subtract (three low registers)*
0	11	1	*Add, subtract (two low registers and immediate)*
0	!= 11	-	MOV (register) - *T2 variant* on page C2-529
1	-	-	*Add, subtract, compare, move (one low register and immediate)* on page C2-327

Add, subtract (three low registers)

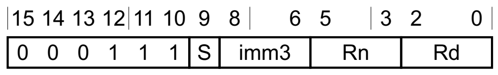
This section describes the encoding of the Add, subtract (three low registers) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page C2-325.



Decode fields	Instruction page
S	
0	ADD (register)
1	SUB (register)

Add, subtract (two low registers and immediate)

This section describes the encoding of the Add, subtract (two low registers and immediate) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page C2-325.



Decode fields	Instruction page
S	
0	ADD (immediate)
1	SUB (immediate)

Add, subtract, compare, move (one low register and immediate)

This section describes the encoding of the Add, subtract, compare, move (one low register and immediate) instruction class. The encodings in this section are decoded from *Shift (immediate), add, subtract, move, and compare* on page C2-325.

15	14	13	12 11	10 8	7 0
0	0	1	op	Rd	imm8

Decode fields op	Instruction page
`00`	MOV (immediate)
`01`	CMP (immediate)
`10`	ADD (immediate)
`11`	SUB (immediate)

C2.2.2 Data-processing (two low registers)

This section describes the encoding of the Data-processing (two low registers) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.

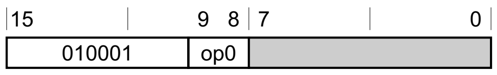
15	14	13	12	11	10	9 6	5 3	2 0
0	1	0	0	0	0	op	Rs	Rd

Decode fields op	Instruction page
`0000`	AND (register)
`0001`	EOR (register)
`0010`	MOV, MOVS (register-shifted register) - *Logical shift left variant* on page C2-533
`0011`	MOV, MOVS (register-shifted register) - *Logical shift right variant* on page C2-533
`0100`	MOV, MOVS (register-shifted register) - *Arithmetic shift right variant* on page C2-533
`0101`	ADC (register)
`0110`	SBC (register)
`0111`	MOV, MOVS (register-shifted register) - *Rotate right variant* on page C2-533
`1000`	TST (register)
`1001`	RSB (immediate)
`1010`	CMP (register)
`1011`	CMN (register)

Decode fields op	Instruction page
1100	ORR (register)
1101	MUL
1110	BIC (register)
1111	MVN (register)

C2.2.3 Special data instructions and branch and exchange

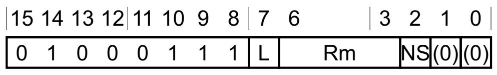
This section describes the encoding of the Special data instructions and branch and exchange group. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.



Decode fields op0	Decode group or instruction page
11	*Branch and exchange*
!= 11	*Add, subtract, compare, move (two high registers)*

Branch and exchange

This section describes the encoding of the Branch and exchange instruction class. The encodings in this section are decoded from *Special data instructions and branch and exchange*.



Decode fields L	Instruction page
0	BX, BXNS
1	BLX, BLXNS

Add, subtract, compare, move (two high registers)

This section describes the encoding of the Add, subtract, compare, move (two high registers) instruction class. The encodings in this section are decoded from *Special data instructions and branch and exchange*.

15	14	13	12	11	10	9 8	7	6 3	2 0
0	1	0	0	0	1	!=11	D	Rs	Rd
						op			

Decode fields			Instruction page
op	D:Rd	Rs	
00	!= 1101	!= 1101	ADD (register)
00	-	1101	ADD (SP plus register) - T1
00	1101	!= 1101	ADD (SP plus register) - T2
01	-	-	CMP (register)
10	-	-	MOV (register)

C2.2.4 Load/store (register offset)

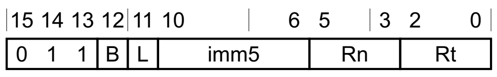
This section describes the encoding of the Load/store (register offset) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.

15	14	13	12	11	10	9	8 6	5 3	2 0
0	1	0	1	L	B	H	Rm	Rn	Rt

Decode fields			Instruction page
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

C2.2.5 Load/store word/byte (immediate offset)

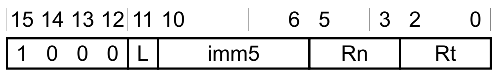
This section describes the encoding of the Load/store word/byte (immediate offset) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.



Decode fields		Instruction page
B	L	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

C2.2.6 Load/store halfword (immediate offset)

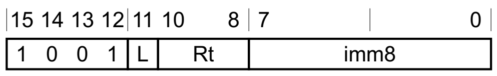
This section describes the encoding of the Load/store halfword (immediate offset) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.



Decode fields	Instruction page
L	
0	STRH (immediate)
1	LDRH (immediate)

C2.2.7 Load/store (SP-relative)

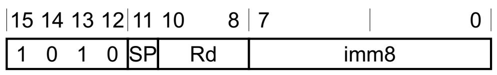
This section describes the encoding of the Load/store (SP-relative) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.



Decode fields	Instruction page
L	
0	STR (immediate)
1	LDR (immediate)

C2.2.8 Add PC/SP (immediate)

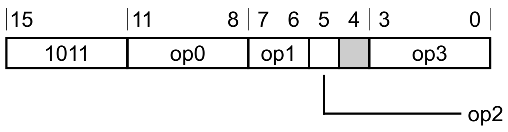
This section describes the encoding of the Add PC/SP (immediate) instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.



Decode fields SP	Instruction page
0	ADR
1	ADD (SP plus immediate)

C2.2.9 Miscellaneous 16-bit instructions

This section describes the encoding of the Miscellaneous 16-bit instructions group. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.

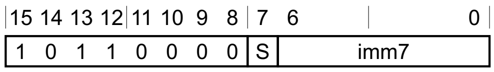


Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0000	-	-	-	*Adjust SP (immediate)* on page C2-332
0010	-	-	-	*Extend* on page C2-332
0110	00	-	-	Unallocated.
0110	01	0	-	Unallocated.
0110	01	1	-	CPS
0110	1x	-	-	Unallocated.
0111	-	-	-	Unallocated.
1000	-	-	-	Unallocated.
1010	10	-	-	Unallocated.
1010	!= 10	-	-	*Reverse bytes* on page C2-332
1110	-	-	-	BKPT
1111	-	-	0000	*Hints* on page C2-333

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
1111	-	-	!= 0000	IT
x0x1	-	-	-	CBNZ, CBZ
x10x	-	-	-	*Push and Pop* on page C2-333

Adjust SP (immediate)

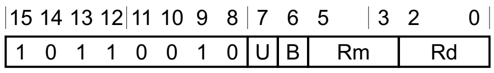
This section describes the encoding of the Adjust SP (immediate) instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page C2-331.



Decode fields	Instruction page
S	
0	ADD (SP plus immediate)
1	SUB (SP minus immediate)

Extend

This section describes the encoding of the Extend instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page C2-331.



Decode fields		Instruction page
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
1	1	UXTB

Reverse bytes

This section describes the encoding of the Reverse bytes instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page C2-331.

15	14	13	12	11	10	9	8	7 6	5 3	2 0
1	0	1	1	1	0	1	0	!=10	Rm	Rd
								op		

Decode fields op	Instruction page
00	REV
01	REV16
11	REVSH

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page C2-331.

15	14	13	12	11	10	9	8	7 4	3	2	1	0
1	0	1	1	1	1	1	1	hint	0	0	0	0

Decode fields hint	Instruction page
0000	NOP
0001	YIELD
0010	WFE
0011	WFI
0100	SEV
0101	Reserved hint, behaves as NOP.
011x	Reserved hint, behaves as NOP.
1xxx	Reserved hint, behaves as NOP.

Push and Pop

This section describes the encoding of the Push and Pop instruction class. The encodings in this section are decoded from *Miscellaneous 16-bit instructions* on page C2-331.

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	L	1	0	P	register_list

Decode fields L	Instruction page
0	STMDB, STMFD
1	LDM, LDMIA, LDMFD

C2.2.10 Load/store multiple

This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.

15	14	13	12	11	10 … 8	7 … 0
1	1	0	0	L	Rn	register_list

Decode fields L	Instruction page
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

C2.2.11 Conditional branch, and Supervisor Call

This section describes the encoding of the Conditional branch, and Supervisor Call group. The encodings in this section are decoded from *16-bit T32 instruction encoding* on page C2-325.

15 … 12	11 … 8	7 … 0
1101	op0	

Decode fields op0	Decode group or instruction page
111x	*Exception generation*
!= 111x	B - *T1 variant* on page C2-404

Exception generation

This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Conditional branch, and Supervisor Call*.

15	14	13	12	11	10	9	8	7 … 0
1	1	0	1	1	1	1	S	imm8

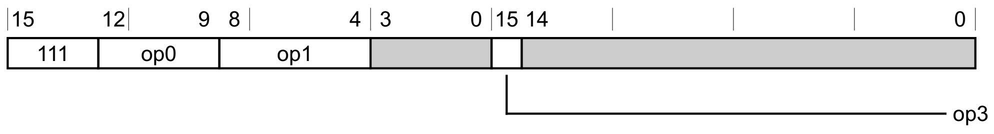
Decode fields S	Instruction page
0	UDF
1	SVC

C2.3 32-bit T32 instruction encoding

This section describes the encoding of the 32-bit T32 instruction encoding group. This section is decoded from *Top level T32 instruction set encoding* on page C2-324.

Note

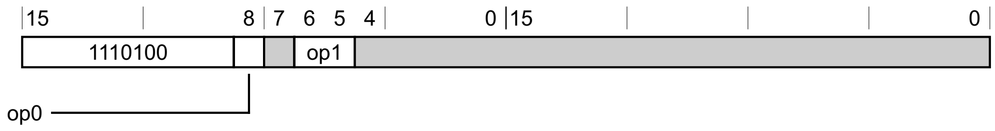
In the decode tables in this section, an entry of - for a field value means the value of the field does not affect the decoding.



Decode fields			Decode group or instruction page
op0	**op1**	**op3**	
`x11x`	-	-	*Coprocessor and floating-point instructions* on page C2-364
`0100`	-	-	*Load/store (multiple, dual, exclusive, acquire-release), table branch*
`0101`	-	-	*Data-processing (shifted register)* on page C2-342
`10xx`	-	1	*Branches and miscellaneous control* on page C2-347
`10x0`	-	0	*Data-processing (modified immediate)* on page C2-344
`10x1`	-	0	*Data-processing (plain binary immediate)* on page C2-345
`1100`	`1xxx0`	-	Unallocated.
`1100`	`!= 1xxx0`	-	*Load/store single* on page C2-349
`1101`	`0xxxx`	-	*Data-processing (register)* on page C2-357
`1101`	`10xxx`	-	*Multiply, multiply accumulate, and absolute difference* on page C2-361
`1101`	`11xxx`	-	*Long multiply and divide* on page C2-363

C2.3.1 Load/store (multiple, dual, exclusive, acquire-release), table branch

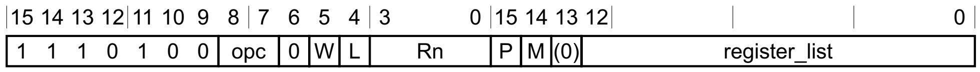
This section describes the encoding of the Load/store (multiple, dual, exclusive, acquire-release), table branch group. The encodings in this section are decoded from *32-bit T32 instruction encoding*.



Decode fields		Decode group or instruction page
op0	op1	
-	0x	*Load/store multiple*
0	10	*Load/store exclusive, load-acquire/store-release, table branch*
0	11	*Load/store dual (post-indexed)* on page C2-340
1	10	*Load/store dual (literal and immediate)* on page C2-340
1	11	*Load/store dual (pre-indexed), secure gateway* on page C2-341

Load/store multiple

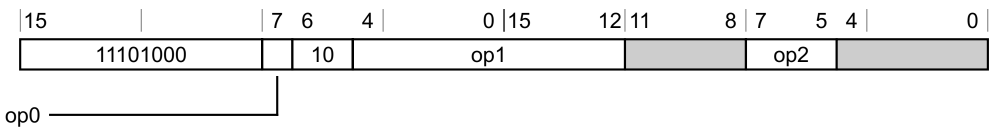
This section describes the encoding of the Load/store multiple instruction class. The encodings in this section are decoded from *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.



Decode fields		Instruction page
opc	L	
00	-	Unallocated.
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	-	Unallocated.

Load/store exclusive, load-acquire/store-release, table branch

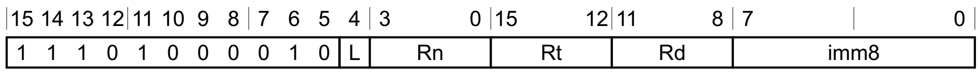
This section describes the encoding of the Load/store exclusive, load-acquire/store-release, table branch group. The encodings in this section are decoded from *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.



Decode fields			Decode group or instruction page
op0	op1	op2	
0	0xxxx1111	-	TT, TTT, TTA, TTAT
0	!= 0xxxx1111	-	*Load/store exclusive*
1	0xxxxxxxx	000	Unallocated.
1	1xxxxxxxx	000	TBB, TBH
1	-	01x	*Load/store exclusive byte/half/dual*
1	-	1xx	*Load-acquire / Store-release* on page C2-339

Load/store exclusive

This section describes the encoding of the Load/store exclusive instruction class. The encodings in this section are decoded from *Load/store exclusive, load-acquire/store-release, table branch* on page C2-337.



Decode fields	Instruction page
L:Rt	
!= 01111	STREX
1xxxx	LDREX

Load/store exclusive byte/half/dual

This section describes the encoding of the Load/store exclusive byte/half/dual instruction class. The encodings in this section are decoded from *Load/store exclusive, load-acquire/store-release, table branch* on page C2-337.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5 4	3 0
1	1	1	0	1	0	0	0	1	1	0	L	Rn	Rt	Rt2	0	1	sz	Rd

Decode fields		**Instruction page**
L	**sz**	
`0`	`00`	STREXB
`0`	`01`	STREXH
`0`	`10`	Unallocated.
`0`	`11`	Unallocated.
`1`	`00`	LDREXB
`1`	`01`	LDREXH
`1`	`10`	Unallocated.
`1`	`11`	Unallocated.

Load-acquire / Store-release

This section describes the encoding of the Load-acquire / Store-release instruction class. The encodings in this section are decoded from *Load/store exclusive, load-acquire/store-release, table branch* on page C2-337.

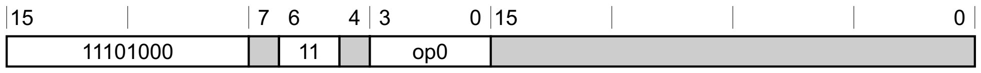
15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5 4	3 0
1	1	1	0	1	0	0	0	1	1	0	L	Rn	Rt	Rt2	1	op	sz	Rd

Decode fields			**Instruction page**
L	**op**	**sz**	
`0`	`0`	`00`	STLB
`0`	`0`	`01`	STLH
`0`	`0`	`10`	STL
`0`	`0`	`11`	Unallocated.
`0`	`1`	`00`	STLEXB
`0`	`1`	`01`	STLEXH
`0`	`1`	`10`	STLEX
`0`	`1`	`11`	Unallocated.
`1`	`0`	`00`	LDAB
`1`	`0`	`01`	LDAH
`1`	`0`	`10`	LDA
`1`	`0`	`11`	Unallocated.

Decode fields			Instruction page
L	op	sz	
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	Unallocated.

Load/store dual (post-indexed)

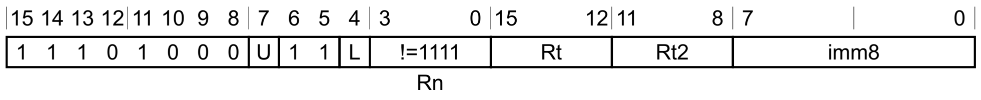
This section describes the encoding of the Load/store dual (post-indexed) group. The encodings in this section are decoded from *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.



Decode fields	Decode group or instruction page
op0	
1111	UNPREDICTABLE
!= 1111	*Load/store dual (immediate, post-indexed)*

Load/store dual (immediate, post-indexed)

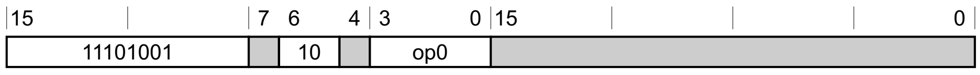
This section describes the encoding of the Load/store dual (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store dual (post-indexed)*.



Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (literal and immediate)

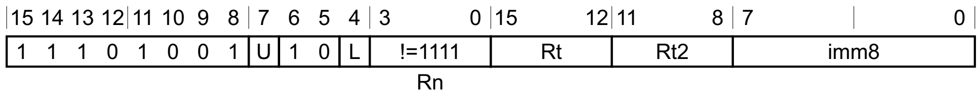
This section describes the encoding of the Load/store dual (literal and immediate) group. The encodings in this section are decoded from *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.



Decode fields op0	Decode group or instruction page
1111	LDRD (literal)
!= 1111	*Load/store dual (immediate)*

Load/store dual (immediate)

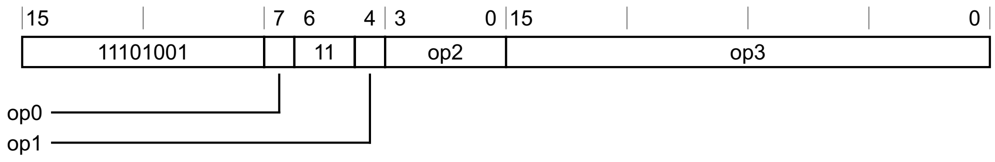
This section describes the encoding of the Load/store dual (immediate) instruction class. The encodings in this section are decoded from *Load/store dual (literal and immediate)* on page C2-340.



Decode fields L	Instruction page
0	STRD (immediate)
1	LDRD (immediate)

Load/store dual (pre-indexed), secure gateway

This section describes the encoding of the Load/store dual (pre-indexed), secure gateway group. The encodings in this section are decoded from *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.



Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	0	1111	-	UNPREDICTABLE
0	1	1111	1110100101111111	SG
0	1	1111	!= 1110100101111111	UNPREDICTABLE

Decode fields				Decode group or instruction page
op0	op1	op2	op3	
1	0	1111	-	UNPREDICTABLE
1	1	1111	-	UNPREDICTABLE
-	-	!= 1111	-	*Load/store dual (immediate, pre-indexed)*

Load/store dual (immediate, pre-indexed)

This section describes the encoding of the Load/store dual (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store dual (pre-indexed), secure gateway* on page C2-341.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	0	1	0	0	1	U	1	1	L	!=1111	Rt	Rt2	imm8
												Rn			

Decode fields	Instruction page
L	
0	STRD (immediate)
1	LDRD (immediate)

C2.3.2 Data-processing (shifted register)

This section describes the encoding of the Data-processing (shifted register) instruction class. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.

15	14	13	12	11	10	9	8 5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	op1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:type	Rd	
0000	0	-	-	-	AND (register) - *AND, rotate right with extend variant* on page C2-394
0000	1	-	!= 0000011	!= 1111	AND (register) - *ANDS, shift or rotate by value variant* on page C2-394
0000	1	-	!= 0000011	1111	TST (register) - *Shift or rotate by value variant* on page C2-718
0000	1	-	0000011	!= 1111	AND (register) - *ANDS, rotate right with extend variant* on page C2-394
0000	1	-	0000011	1111	TST (register) - *Rotate right with extend variant* on page C2-718
0001	-	-	-	-	BIC (register)
0010	0	!= 1111	-	-	ORR (register) - *ORR, rotate right with extend variant* on page C2-560
0010	0	1111	-	-	MOV (register) - *MOV, rotate right with extend variant* on page C2-530

Decode fields					Instruction page
op1	S	Rn	imm3:imm2:type	Rd	
0010	1	!= 1111	-	-	ORR (register) - *ORRS, rotate right with extend variant* on page C2-560
0010	1	1111	-	-	MOV (register) - *MOVS, rotate right with extend variant* on page C2-530
0011	0	!= 1111	-	-	ORN (register) - *ORN, rotate right with extend variant* on page C2-557
0011	0	1111	-	-	MVN (register) - *MVN, rotate right with extend variant* on page C2-553
0011	1	!= 1111	-	-	ORN (register) - *ORNS, rotate right with extend variant* on page C2-557
0011	1	1111	-	-	MVN (register) - *MVNS, rotate right with extend variant* on page C2-553
0100	0	-	-	-	EOR (register) - *EOR, rotate right with extend variant* on page C2-434
0100	1	-	!= 0000011	!= 1111	EOR (register) - *EORS, shift or rotate by value variant* on page C2-434
0100	1	-	!= 0000011	1111	TEQ (register) - *Shift or rotate by value variant* on page C2-715
0100	1	-	0000011	!= 1111	EOR (register) - *EORS, rotate right with extend variant* on page C2-434
0100	1	-	0000011	1111	TEQ (register) - *Rotate right with extend variant* on page C2-715
0101	-	-	-	-	Unallocated.
0110	0	-	xxxxx00	-	PKHBT, PKHTB - *PKHBT variant* on page C2-562
0110	0	-	xxxxx01	-	Unallocated.
0110	0	-	xxxxx10	-	PKHBT, PKHTB - *PKHTB variant* on page C2-562
0110	0	-	xxxxx11	-	Unallocated.
0111	-	-	-	-	Unallocated.
1000	0	!= 1101	-	-	ADD (register) - *ADD, rotate right with extend variant* on page C2-388
1000	0	1101	-	-	ADD (SP plus register) - *ADD, rotate right with extend variant* on page C2-380
1000	1	!= 1101	-	!= 1111	ADD (register) - *ADDS, rotate right with extend variant* on page C2-389
1000	1	1101	-	!= 1111	ADD (SP plus register) - *ADDS, rotate right with extend variant* on page C2-381
1000	1	-	-	1111	CMN (register)
1001	-	-	-	-	Unallocated.
1010	-	-	-	-	ADC (register)
1011	-	-	-	-	SBC (register)
1100	-	-	-	-	Unallocated.
1101	0	!= 1101	-	-	SUB (register) - *SUB, rotate right with extend variant* on page C2-702
1101	0	1101	-	-	SUB (SP minus register) - *SUB, rotate right with extend variant* on page C2-696

Decode fields op1	S	Rn	imm3:imm2:type	Rd	Instruction page
1101	1	!= 1101	-	!= 1111	SUB (register) - *SUBS, rotate right with extend variant* on page C2-702
1101	1	1101	-	!= 1111	SUB (SP minus register) - *SUBS, rotate right with extend variant* on page C2-696
1101	1	-	-	1111	CMP (register)
1110	-	-	-	-	RSB (register)
1111	-	-	-	-	Unallocated.

C2.3.3 Data-processing (modified immediate)

This section describes the encoding of the Data-processing (modified immediate) instruction class. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.

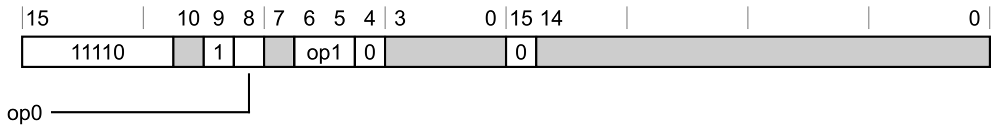
15 14 13 12	11	10	9	8 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1	0	i	0	op1	S	Rn	0	imm3	Rd	imm8

Decode fields op1	S	Rn	Rd	Instruction page
0000	0	-	-	AND (immediate) - *AND variant* on page C2-393
0000	1	-	!= 1111	AND (immediate) - *ANDS variant* on page C2-393
0000	1	-	1111	TST (immediate)
0001	-	-	-	BIC (immediate)
0010	0	!= 1111	-	ORR (immediate) - *ORR variant* on page C2-559
0010	0	1111	-	MOV (immediate) - *MOV variant* on page C2-527
0010	1	!= 1111	-	ORR (immediate) - *ORRS variant* on page C2-559
0010	1	1111	-	MOV (immediate) - *MOVS variant* on page C2-527
0011	0	!= 1111	-	ORN (immediate) - *Non flag setting variant* on page C2-556
0011	0	1111	-	MVN (immediate) - *MVN variant* on page C2-552
0011	1	!= 1111	-	ORN (immediate) - *Flag setting variant* on page C2-556
0011	1	1111	-	MVN (immediate) - *MVNS variant* on page C2-552
0100	0	-	-	EOR (immediate) - *EOR variant* on page C2-433
0100	1	-	!= 1111	EOR (immediate) - *EORS variant* on page C2-433
0100	1	-	1111	TEQ (immediate)
0101	-	-	-	Unallocated.
011x	-	-	-	Unallocated.

Decode fields				Instruction page
op1	S	Rn	Rd	
`1000`	`0`	`!= 1101`	`-`	ADD (immediate) - *ADD variant* on page C2-383
`1000`	`0`	`1101`	`-`	ADD (SP plus immediate) - *ADD variant* on page C2-378
`1000`	`1`	`!= 1101`	`!= 1111`	ADD (immediate) - *ADDS variant* on page C2-384
`1000`	`1`	`1101`	`!= 1111`	ADD (SP plus immediate) - *ADDS variant* on page C2-378
`1000`	`1`	`-`	`1111`	CMN (immediate)
`1001`	`-`	`-`	`-`	Unallocated.
`1010`	`-`	`-`	`-`	ADC (immediate)
`1011`	`-`	`-`	`-`	SBC (immediate)
`1100`	`-`	`-`	`-`	Unallocated.
`1101`	`0`	`!= 1101`	`-`	SUB (immediate) - *SUB variant* on page C2-698
`1101`	`0`	`1101`	`-`	SUB (SP minus immediate) - *SUB variant* on page C2-694
`1101`	`1`	`!= 1101`	`!= 1111`	SUB (immediate) - *SUBS variant* on page C2-699
`1101`	`1`	`1101`	`!= 1111`	SUB (SP minus immediate) - *SUBS variant* on page C2-694
`1101`	`1`	`-`	`1111`	CMP (immediate)
`1110`	`-`	`-`	`-`	RSB (immediate)
`1111`	`-`	`-`	`-`	Unallocated.

C2.3.4 Data-processing (plain binary immediate)

This section describes the encoding of the Data-processing (plain binary immediate) group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.



Decode fields		Decode group or instruction page
op0	op1	
`0`	`0x`	*Data-processing (simple immediate)* on page C2-346
`0`	`10`	*Move Wide (16-bit immediate)* on page C2-346
`0`	`11`	Unallocated.
`1`	`-`	*Saturate, Bitfield* on page C2-346

Data-processing (simple immediate)

This section describes the encoding of the Data-processing (simple immediate) instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)* on page C2-345.

15 14 13 12 11	10	9 8	7	6	5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	o1	0	o2	0	Rn	0	imm3	Rd	imm8

Decode fields			Instruction page
o1	o2	Rn	
0	0	!= 11x1	ADD (immediate)
0	0	1101	ADD (SP plus immediate)
0	0	1111	ADR - T3
0	1	-	Unallocated.
1	0	-	Unallocated.
1	1	!= 11x1	SUB (immediate)
1	1	1101	SUB (SP minus immediate)
1	1	1111	ADR - T2

Move Wide (16-bit immediate)

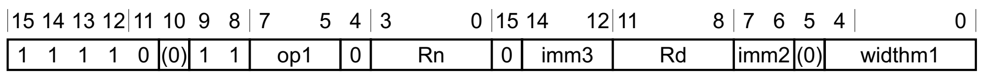
This section describes the encoding of the Move Wide (16-bit immediate) instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)* on page C2-345.

15 14 13 12 11	10	9 8	7	6 5 4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	o1	1 0 0	imm4	0	imm3	Rd	imm8

Decode fields	Instruction page
o1	
0	MOV (immediate)
1	MOVT

Saturate, Bitfield

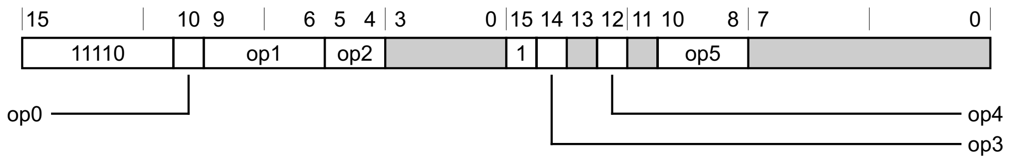
This section describes the encoding of the Saturate, Bitfield instruction class. The encodings in this section are decoded from *Data-processing (plain binary immediate)* on page C2-345.



Decode fields			Instruction page
op1	Rn	imm3:imm2	
000	-	-	SSAT - *Logical shift left variant* on page C2-644
001	-	!= 00000	SSAT - *Arithmetic shift right variant* on page C2-644
001	-	00000	SSAT16
010	-	-	SBFX
011	!= 1111	-	BFI
011	1111	-	BFC
100	-	-	USAT - *Logical shift left variant* on page C2-745
101	-	!= 00000	USAT - *Arithmetic shift right variant* on page C2-745
101	-	00000	USAT16
110	-	-	UBFX
111	-	-	Unallocated.

C2.3.5 Branches and miscellaneous control

This section describes the encoding of the Branches and miscellaneous control group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.



Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
0	1110	0x	0	0	-	MSR (register)
0	1110	10	0	0	000	*Hints* on page C2-348
0	1110	10	0	0	!= 000	Unallocated.
0	1110	11	0	0	-	*Miscellaneous system* on page C2-349
0	1111	0x	0	0	-	Unallocated.
0	1111	1x	0	0	-	MRS
1	1110	-	0	0	-	Unallocated.

Decode fields						Decode group or instruction page
op0	op1	op2	op3	op4	op5	
1	1111	0x	0	0	-	Unallocated.
1	1111	1x	0	0	-	*Exception generation* on page C2-349
-	!= 111x	-	0	0	-	B - *T3 variant* on page C2-404
-	-	-	0	1	-	B - *T4 variant* on page C2-405
-	-	-	1	0	-	Unallocated.
-	-	-	1	1	-	BL

Hints

This section describes the encoding of the Hints instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page C2-347.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7 4	3 0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	hint	option

Decode fields		Instruction page
hint	option	
0000	0000	NOP
0000	0001	YIELD
0000	0010	WFE
0000	0011	WFI
0000	0100	SEV
0000	0101	Reserved hint, behaves as NOP.
0000	011x	Reserved hint, behaves as NOP.
0000	1xxx	Reserved hint, behaves as NOP.
0001	-	Reserved hint, behaves as NOP.
001x	-	Reserved hint, behaves as NOP.
01xx	-	Reserved hint, behaves as NOP.
10xx	-	Reserved hint, behaves as NOP.
110x	-	Reserved hint, behaves as NOP.
1110	-	Reserved hint, behaves as NOP.
1111	-	DBG

Miscellaneous system

This section describes the encoding of the Miscellaneous system instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page C2-347.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7 4	3 0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	opc	option

Decode fields opc	Instruction page
000x	Unallocated.
0010	CLREX
0011	Unallocated.
0100	DSB
0101	DMB
0110	ISB
0111	Unallocated.
1xxx	Unallocated.

Exception generation

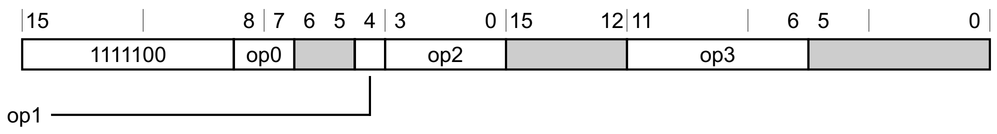
This section describes the encoding of the Exception generation instruction class. The encodings in this section are decoded from *Branches and miscellaneous control* on page C2-347.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 0
1	1	1	1	0	1	1	1	1	1	1	o1	imm4	1	0	o2	0	imm12

Decode fields		Instruction page
o1	o2	
0	0	Unallocated.
0	1	Unallocated.
1	0	Unallocated.
1	1	UDF

C2.3.6 Load/store single

This section describes the encoding of the Load/store single group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.



Decode fields				Decode group or instruction page
op0	op1	op2	op3	
00	-	!= 1111	000000	*Load/store, unsigned (register offset)* on page C2-351
00	-	!= 1111	000001	Unallocated.
00	-	!= 1111	00001x	Unallocated.
00	-	!= 1111	0001xx	Unallocated.
00	-	!= 1111	001xxx	Unallocated.
00	-	!= 1111	01xxxx	Unallocated.
00	-	!= 1111	10x0xx	Unallocated.
00	-	!= 1111	10x1xx	*Load/store, unsigned (immediate, post-indexed)* on page C2-351
00	-	!= 1111	1100xx	*Load/store, unsigned (negative immediate)* on page C2-352
00	-	!= 1111	1110xx	*Load/store, unsigned (unprivileged)* on page C2-352
00	-	!= 1111	11x1xx	*Load/store, unsigned (immediate, pre-indexed)* on page C2-353
01	-	!= 1111	-	*Load/store, unsigned (positive immediate)* on page C2-353
0x	-	1111	-	*Load, unsigned (literal)* on page C2-354
10	1	!= 1111	000000	*Load/store, signed (register offset)* on page C2-354
10	1	!= 1111	000001	Unallocated.
10	1	!= 1111	00001x	Unallocated.
10	1	!= 1111	0001xx	Unallocated.
10	1	!= 1111	001xxx	Unallocated.
10	1	!= 1111	01xxxx	Unallocated.
10	1	!= 1111	10x0xx	Unallocated.
10	1	!= 1111	10x1xx	*Load/store, signed (immediate, post-indexed)* on page C2-354
10	1	!= 1111	1100xx	*Load/store, signed (negative immediate)* on page C2-355
10	1	!= 1111	1110xx	*Load/store, signed (unprivileged)* on page C2-355
10	1	!= 1111	11x1xx	*Load/store, signed (immediate, pre-indexed)* on page C2-356
11	1	!= 1111	-	*Load/store, signed (positive immediate)* on page C2-356
1x	1	1111	-	*Load, signed (literal)* on page C2-357

Load/store, unsigned (register offset)

This section describes the encoding of the Load/store, unsigned (register offset) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8 7 6	5 4	3 0
1 1 1 1 1 0 0 0 0	size	L	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD (register) - *Preload read variant* on page C2-565
01	0	-	STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD (register) - *Preload write variant* on page C2-565
10	0	-	STR (register)
10	1	-	LDR (register)
11	-	-	Unallocated.

Load/store, unsigned (immediate, post-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11	10	9	8	7 0
1 1 1 1 1 0 0 0 0	size	L	!=1111	Rt	1	0	U	1	imm8
			Rn						

Decode fields		Instruction page
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	-	Unallocated.

Load/store, unsigned (negative immediate)

This section describes the encoding of the Load/store, unsigned (negative immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	size	L	!=1111	Rt	1	1	0	0	imm8
											Rn						

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - *Preload read variant* on page C2-566
01	0	-	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - *Preload write variant* on page C2-566
10	0	-	STR (immediate)
10	1	-	LDR (immediate)
11	-	-	Unallocated.

Load/store, unsigned (unprivileged)

This section describes the encoding of the Load/store, unsigned (unprivileged) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	size	L	!=1111	Rt	1	1	1	0	imm8
											Rn						

Decode fields		Instruction page
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11	-	Unallocated.

Load/store, unsigned (immediate, pre-indexed)

This section describes the encoding of the Load/store, unsigned (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	size	L	!=1111	Rt	1	1	U	1	imm8
											Rn						

Decode fields		Instruction page
size	L	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11	-	Unallocated.

Load/store, unsigned (positive immediate)

This section describes the encoding of the Load/store, unsigned (positive immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11 0
1	1	1	1	1	0	0	0	1	size	L	!=1111	Rt	imm12
											Rn		

Decode fields			Instruction page
size	L	Rt	
00	0	-	STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) - *Preload read variant* on page C2-566
01	0	-	STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) - *Preload write variant* on page C2-566
10	0	-	STR (immediate)
10	1	-	LDR (immediate)

Load, unsigned (literal)

This section describes the encoding of the Load, unsigned (literal) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3	2	1	0	15 12	11 0
1	1	1	1	1	0	0	0	U	size	L	1	1	1	1	Rt	imm12

Decode fields			Instruction page
size	L	Rt	
00	1	!= 1111	LDRB (literal)
00	1	1111	PLD (literal)
01	1	!= 1111	LDRH (literal)
10	1	-	LDR (literal)
11	-	-	Unallocated.

Load/store, signed (register offset)

This section describes the encoding of the Load/store, signed (register offset) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7	6	5 4	3 0
1	1	1	1	1	0	0	1	0	size	1	!=1111	Rt	0	0	0	0	0	0	imm2	Rm
											Rn									

Decode fields		Instruction page
size	Rt	
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP.
1x	-	Unallocated.

Load/store, signed (immediate, post-indexed)

This section describes the encoding of the Load/store, signed (immediate, post-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	1	0	size	1	!=1111	Rt	1	0	U	1	imm8
											Rn						

Decode fields size	Instruction page
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	Unallocated.

Load/store, signed (negative immediate)

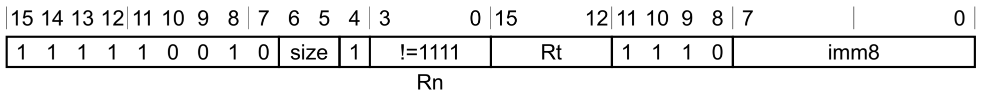
This section describes the encoding of the Load/store, signed (negative immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	1	0	size	1	!=1111	Rt	1	1	0	0	imm8
											Rn						

Decode fields		Instruction page
size	Rt	
00	-	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP.
1x	-	Unallocated.

Load/store, signed (unprivileged)

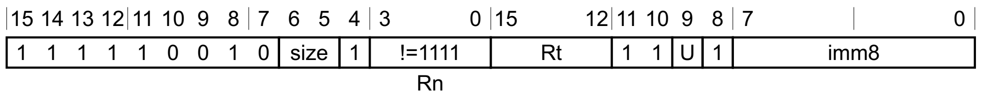
This section describes the encoding of the Load/store, signed (unprivileged) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.



Decode fields size	Instruction page
`00`	LDRSBT
`01`	LDRSHT
1x	Unallocated.

Load/store, signed (immediate, pre-indexed)

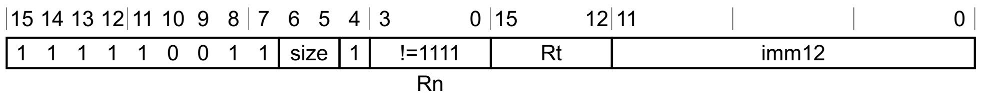
This section describes the encoding of the Load/store, signed (immediate, pre-indexed) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.



Decode fields size	Instruction page
`00`	LDRSB (immediate)
`01`	LDRSH (immediate)
1x	Unallocated.

Load/store, signed (positive immediate)

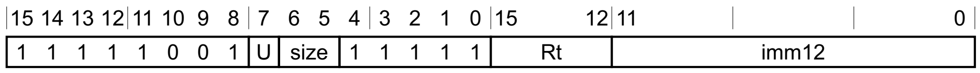
This section describes the encoding of the Load/store, signed (positive immediate) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.



Decode fields size	 Rt	Instruction page
`00`	!= 1111	LDRSB (immediate)
`00`	1111	PLI (immediate, literal)
`01`	!= 1111	LDRSH (immediate)
`01`	1111	Reserved hint, behaves as NOP.

Load, signed (literal)

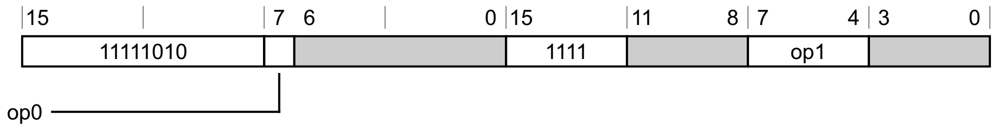
This section describes the encoding of the Load, signed (literal) instruction class. The encodings in this section are decoded from *Load/store single* on page C2-349.



Decode fields		Instruction page
size	Rt	
`00`	`!= 1111`	LDRSB (literal)
`00`	`1111`	PLI (immediate, literal)
`01`	`!= 1111`	LDRSH (literal)
`01`	`1111`	Reserved hint, behaves as NOP.
`1x`	-	Unallocated.

C2.3.7 Data-processing (register)

This section describes the encoding of the Data-processing (register) group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.



Decode fields		Decode group or instruction page
op0	op1	
`0`	`0000`	MOV, MOVS (register-shifted register) - *Flag setting variant* on page C2-533
`0`	`0001`	Unallocated.
`0`	`001x`	Unallocated.
`0`	`01xx`	Unallocated.
`0`	`1xxx`	*Register extends*
`1`	`0xxx`	*Parallel add-subtract* on page C2-358
`1`	`10xx`	*Data-processing (two source registers)* on page C2-360
`1`	`11xx`	Unallocated.

Register extends

This section describes the encoding of the Register extends instruction class. The encodings in this section are decoded from *Data-processing (register)*.

15	14	13	12	11	10	9	8	7	6 5	4	3 0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	op1	U	Rn	1	1	1	1	Rd	1	(0)	rotate	Rm

Decode fields			Instruction page
op1	**U**	**Rn**	
`00`	`0`	!= 1111	SXTAH
`00`	`0`	1111	SXTH
`00`	1	!= 1111	UXTAH
`00`	1	1111	UXTH
`01`	`0`	!= 1111	SXTAB16
`01`	`0`	1111	SXTB16
`01`	1	!= 1111	UXTAB16
`01`	1	1111	UXTB16
10	`0`	!= 1111	SXTAB
10	`0`	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11	-	-	Unallocated.

Parallel add-subtract

This section describes the encoding of the Parallel add-subtract instruction class. The encodings in this section are decoded from *Data-processing (register)* on page C2-357.

15	14	13	12	11	10	9	8	7	6 4	3 0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	0	1	op1	Rn	1	1	1	1	Rd	0	U	H	S	Rm

Decode fields				Instruction page
op1	**U**	**H**	**S**	
`000`	`0`	`0`	`0`	SADD8
`000`	`0`	`0`	1	QADD8
`000`	`0`	1	`0`	SHADD8
`000`	`0`	1	1	Unallocated.
`000`	1	`0`	`0`	UADD8
`000`	1	`0`	1	UQADD8

Decode fields				Instruction page
op1	U	H	S	
000	1	1	0	UHADD8
000	1	1	1	Unallocated.
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	Unallocated.
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	Unallocated.
010	0	0	0	SASX
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	Unallocated.
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	Unallocated.
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	Unallocated.
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	Unallocated.
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	Unallocated.
101	1	0	0	USUB16
101	1	0	1	UQSUB16

Decode fields				Instruction page
op1	U	H	S	
101	1	1	0	UHSUB16
101	1	1	1	Unallocated.
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	Unallocated.
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	Unallocated.
111	-	-	-	Unallocated.

Data-processing (two source registers)

This section describes the encoding of the Data-processing (two source registers) instruction class. The encodings in this section are decoded from *Data-processing (register)* on page C2-357.

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

Decode fields		Instruction page
op1	op2	
000	00	QADD
000	01	QDADD
000	10	QSUB
000	11	QDSUB
001	00	REV
001	01	REV16
001	10	RBIT
001	11	REVSH
010	00	SEL
010	01	Unallocated.
010	1x	Unallocated.
011	00	CLZ

Decode fields		Instruction page
op1	**op2**	
011	01	Unallocated.
011	1x	Unallocated.
1xx	-	Unallocated.

C2.3.8 Multiply, multiply accumulate, and absolute difference

This section describes the encoding of the Multiply, multiply accumulate, and absolute difference group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.

15 … 7	6 … 0 | 15 … 8	7 6	5 … 0
111110110		op0	

Decode fields	Decode group or instruction page
op0	
00	*Multiply and absolute difference*
01	Unallocated.
1x	Unallocated.

Multiply and absolute difference

This section describes the encoding of the Multiply and absolute difference instruction class. The encodings in this section are decoded from *Multiply, multiply accumulate, and absolute difference*.

15	14	13	12	11	10	9	8	7	6 … 4	3 … 0	15 … 12	11 … 8	7	6	5 4	3 … 0
1	1	1	1	1	0	1	1	0	op1	Rn	Ra	Rd	0	0	op2	Rm

Decode fields			Instruction page
op1	**Ra**	**op2**	
000	!= 1111	00	MLA
000	-	01	MLS
000	-	1x	Unallocated.
000	1111	00	MUL
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT - *SMLABB variant* on page C2-623
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT - *SMLABT variant* on page C2-623
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT - *SMLATB variant* on page C2-623
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT - *SMLATT variant* on page C2-623

Decode fields			Instruction page
op1	Ra	op2	
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT - *SMULBB variant* on page C2-639
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT - *SMULBT variant* on page C2-639
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT - *SMULTB variant* on page C2-639
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT - *SMULTT variant* on page C2-639
010	!= 1111	00	SMLAD, SMLADX - *SMLAD variant* on page C2-625
010	!= 1111	01	SMLAD, SMLADX - *SMLADX variant* on page C2-625
010	-	1x	Unallocated.
010	1111	00	SMUAD, SMUADX - *SMUAD variant* on page C2-638
010	1111	01	SMUAD, SMUADX - *SMUADX variant* on page C2-638
011	!= 1111	00	SMLAWB, SMLAWT - *SMLAWB variant* on page C2-631
011	!= 1111	01	SMLAWB, SMLAWT - *SMLAWT variant* on page C2-631
011	-	1x	Unallocated.
011	1111	00	SMULWB, SMULWT - *SMULWB variant* on page C2-642
011	1111	01	SMULWB, SMULWT - *SMULWT variant* on page C2-642
100	!= 1111	00	SMLSD, SMLSDX - *SMLSD variant* on page C2-632
100	!= 1111	01	SMLSD, SMLSDX - *SMLSDX variant* on page C2-632
100	-	1x	Unallocated.
100	1111	00	SMUSD, SMUSDX - *SMUSD variant* on page C2-643
100	1111	01	SMUSD, SMUSDX - *SMUSDX variant* on page C2-643
101	!= 1111	00	SMMLA, SMMLAR - *SMMLA variant* on page C2-635
101	!= 1111	01	SMMLA, SMMLAR - *SMMLAR variant* on page C2-635
101	-	1x	Unallocated.
101	1111	00	SMMUL, SMMULR - *SMMUL variant* on page C2-637
101	1111	01	SMMUL, SMMULR - *SMMULR variant* on page C2-637
110	-	00	SMMLS, SMMLSR - *SMMLS variant* on page C2-636
110	-	01	SMMLS, SMMLSR - *SMMLSR variant* on page C2-636
110	-	1x	Unallocated.
111	!= 1111	00	USADA8
111	-	01	Unallocated.
111	-	1x	Unallocated.
111	1111	00	USAD8

C2.3.9 Long multiply and divide

This section describes the encoding of the Long multiply and divide instruction class. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.

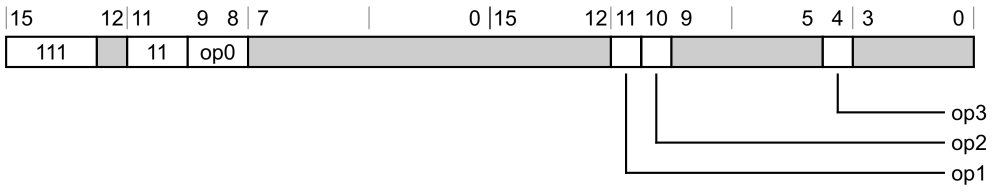
15 14 13 12 11 10 9 8	7 6 4	3 0	15 12	11 8	7 4	3 0
1 1 1 1 1 0 1 1 1	op1	Rn	RdLo	RdHi	op2	Rm

Decode fields		Instruction page
op1	op2	
000	!= 0000	Unallocated.
000	0000	SMULL
001	!= 1111	Unallocated.
001	1111	SDIV
010	!= 0000	Unallocated.
010	0000	UMULL
011	!= 1111	Unallocated.
011	1111	UDIV
100	0000	SMLAL
100	0001	Unallocated.
100	001x	Unallocated.
100	01xx	Unallocated.
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT - *SMLALBB variant* on page C2-627
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT - *SMLALBT variant* on page C2-627
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT - *SMLALTB variant* on page C2-627
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT - *SMLALTT variant* on page C2-627
100	1100	SMLALD, SMLALDX - *SMLALD variant* on page C2-629
100	1101	SMLALD, SMLALDX - *SMLALDX variant* on page C2-629
100	111x	Unallocated.
101	0xxx	Unallocated.
101	10xx	Unallocated.
101	1100	SMLSLD, SMLSLDX - *SMLSLD variant* on page C2-633
101	1101	SMLSLD, SMLSLDX - *SMLSLDX variant* on page C2-633
101	111x	Unallocated.
110	0000	UMLAL
110	0001	Unallocated.

Decode fields		Instruction page
op1	op2	
110	001x	Unallocated.
110	010x	Unallocated.
110	0110	UMAAL
110	0111	Unallocated.
110	1xxx	Unallocated.
111	-	Unallocated.

C2.3.10 Coprocessor and floating-point instructions

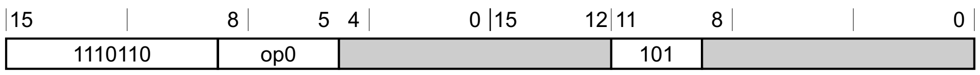
This section describes the encoding of the Coprocessor and floating-point instructions group. The encodings in this section are decoded from *32-bit T32 instruction encoding* on page C2-336.



Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0x	1	0	-	*Floating-point load/store and 64-bit register moves*
0x	1	1	-	Unallocated.
10	1	0	0	*Floating-point data-processing* on page C2-366
10	1	0	1	*Floating-point 32-bit register moves* on page C2-370
10	1	1	0	Unallocated.
10	1	1	1	Unallocated.
11	-	-	-	Unallocated.
!= 11	!= 1	-	-	*Coprocessor* on page C2-371

Floating-point load/store and 64-bit register moves

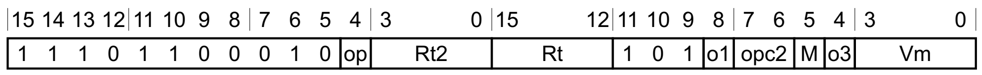
This section describes the encoding of the Floating-point load/store and 64-bit register moves group. The encodings in this section are decoded from *Coprocessor and floating-point instructions*.



Decode fields op0	Decode group or instruction page
`0000`	Unallocated.
`0010`	*Floating-point 64-bit move*
`!= 00x0`	*Floating-point load/store*

Floating-point 64-bit move

This section describes the encoding of the Floating-point 64-bit move instruction class. The encodings in this section are decoded from *Floating-point load/store and 64-bit register moves* on page C2-364.



Decode fields				Instruction page
op	o1	opc2	o3	
-	-	`!= 00`	-	Unallocated.
-	-	-	`0`	Unallocated.
`0`	`0`	`00`	`1`	VMOV (between two general-purpose registers and two single-precision registers)
`0`	`1`	`00`	`1`	VMOV (between two general-purpose registers and a doubleword register)
`1`	`0`	`00`	`1`	VMOV (between two general-purpose registers and two single-precision registers)
`1`	`1`	`00`	`1`	VMOV (between two general-purpose registers and a doubleword register)

Floating-point load/store

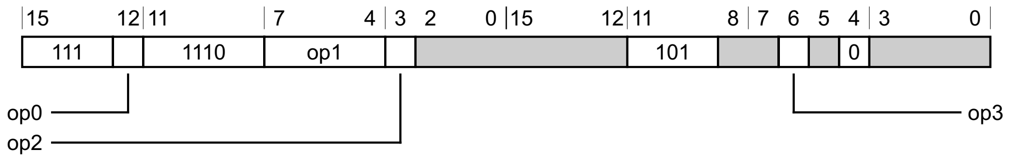
This section describes the encoding of the Floating-point load/store instruction class. The encodings in this section are decoded from *Floating-point load/store and 64-bit register moves* on page C2-364.

15 14 13 12 11 10 9	8	7	6	5	4	3 0	15 12	11 10 9	8	7 0
1 1 1 0 1 1 0	P	U	D	W	L	Rn	Vd	1 0 1	sz	imm8

Decode fields						Instruction page
P	U	W	L	sz	imm8	
0	0	1	0	0	-	VLSTM
0	0	1	-	1	-	Unallocated.
0	0	1	1	0	-	VLLDM
0	1	-	0	0	-	VSTM
0	1	-	0	1	xxxxxxx0	VSTM
0	1	-	0	1	xxxxxxx1	FSTMDBX, FSTMIAX - *Increment After variant* on page C2-438
0	1	-	1	0	-	VLDM
0	1	-	1	1	xxxxxxx0	VLDM
0	1	-	1	1	xxxxxxx1	FLDMDBX, FLDMIAX - *Increment After variant* on page C2-436
1	-	0	0	-	-	VSTR
1	-	0	1	-	-	VLDR
1	0	1	0	0	-	VSTM
1	0	1	0	1	xxxxxxx0	VSTM
1	0	1	0	1	xxxxxxx1	FSTMDBX, FSTMIAX - *Decrement Before variant* on page C2-438
1	0	1	1	0	-	VLDM
1	0	1	1	1	xxxxxxx0	VLDM
1	0	1	1	1	xxxxxxx1	FLDMDBX, FLDMIAX - *Decrement Before variant* on page C2-436
1	1	1	-	-	-	Unallocated.

Floating-point data-processing

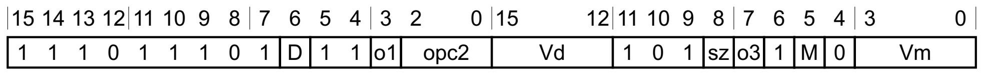
This section describes the encoding of the Floating-point data-processing group. The encodings in this section are decoded from *Coprocessor and floating-point instructions* on page C2-364.



Decode fields				Decode group or instruction page
op0	op1	op2	op3	
0	1x11	-	1	*Floating-point data-processing (two registers)*
0	1x11	-	0	VMOV (immediate)
0	!= 1x11	-	-	*Floating-point data-processing (three registers)* on page C2-368
1	0xxx	-	0	VSEL
1	0xxx	-	1	Unallocated.
1	1x00	-	-	*Floating-point minNum / maxNum* on page C2-369
1	1x01	-	-	Unallocated.
1	1x10	-	-	Unallocated.
1	1x11	0	-	Unallocated.
1	1x11	1	0	Unallocated.
1	1x11	1	1	*Floating-point directed convert to integer* on page C2-369

Floating-point data-processing (two registers)

This section describes the encoding of the Floating-point data-processing (two registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page C2-366.



Decode fields			Instruction page
o1	opc2	o3	
0	000	0	VMOV (register)
0	000	1	VABS
0	001	0	VNEG
0	001	1	VSQRT
0	010	0	VCVTB
0	010	1	VCVTT
0	011	0	VCVTB

Decode fields			Instruction page
o1	opc2	o3	
0	011	1	VCVTT
0	100	0	VCMP - T1
0	100	1	VCMPE - T1
0	101	0	VCMP - T2
0	101	1	VCMPE - T2
0	110	0	VRINTR
0	110	1	VRINTZ
0	111	0	VRINTX
0	111	1	VCVT (between double-precision and single-precision)
1	000	-	VCVT (integer to floating-point)
1	001	-	Unallocated.
1	01x	-	VCVT (between floating-point and fixed-point)
1	100	0	VCVTR
1	100	1	VCVT (floating-point to integer)
1	101	0	VCVTR
1	101	1	VCVT (floating-point to integer)
1	11x	-	VCVT (between floating-point and fixed-point)

Floating-point data-processing (three registers)

This section describes the encoding of the Floating-point data-processing (three registers) instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page C2-366.

15	14	13	12	11	10	9	8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	o0	D	o1	Vn	Vd	1	0	1	sz	N	o2	M	0	Vm

Decode fields			Instruction page
o0	o1	o2	
0	00	0	VMLA
0	00	1	VMLS
0	01	0	VNMLS
0	01	1	VNMLA
0	10	0	VMUL
0	10	1	VNMUL

Decode fields o0	o1	o2	Instruction page
0	11	0	VADD
0	11	1	VSUB
1	00	0	VDIV
1	01	0	VFNMS
1	01	1	VFNMA
1	10	0	VFMA
1	10	1	VFMS

Floating-point minNum / maxNum

This section describes the encoding of the Floating-point minNum / maxNum instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page C2-366.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	1	sz	N	op	M	0	Vm

Decode fields op	Instruction page
0	VMAXNM
1	VMINNM

Floating-point directed convert to integer

This section describes the encoding of the Floating-point directed convert to integer instruction class. The encodings in this section are decoded from *Floating-point data-processing* on page C2-366.

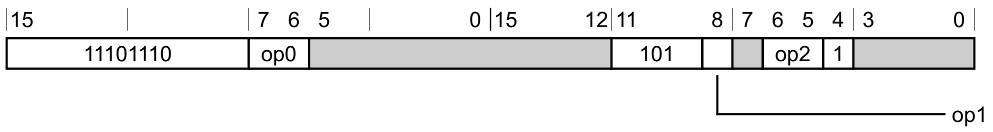
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	1	1	1	o1	rm	Vd	1	0	1	sz	nU	1	M	0	Vm

Decode fields o1	rm	Instruction page
0	00	VRINTA
0	01	VRINTN
0	10	VRINTP
0	11	VRINTM
1	00	VCVTA

Decode fields		Instruction page
o1	rm	
1	01	VCVTN
1	10	VCVTP
1	11	VCVTM

Floating-point 32-bit register moves

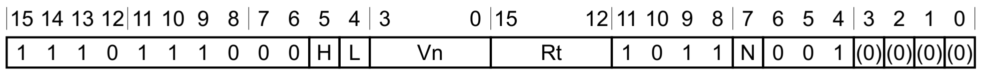
This section describes the encoding of the Floating-point 32-bit register moves group. The encodings in this section are decoded from *Coprocessor and floating-point instructions* on page C2-364.



Decode fields			Decode group or instruction page
op0	op1	op2	
00	1	00	*Floating-point 32-bit move doubleword*
00	1	!= 00	Unallocated.
!= 00	1	-	Unallocated.
-	0	-	*Floating-point 32-bit move*

Floating-point 32-bit move doubleword

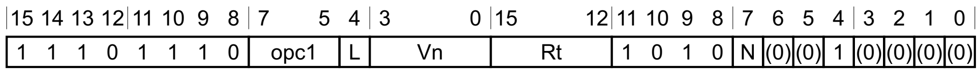
This section describes the encoding of the Floating-point 32-bit move doubleword instruction class. The encodings in this section are decoded from *Floating-point 32-bit register moves*.



Decode fields	Instruction page
L	
0	VMOV (single general-purpose register to half of doubleword register)
1	VMOV (half of doubleword register to single general-purpose register)

Floating-point 32-bit move

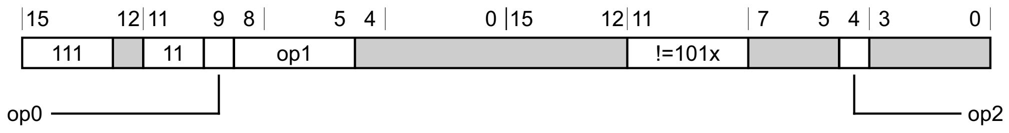
This section describes the encoding of the Floating-point 32-bit move instruction class. The encodings in this section are decoded from *Floating-point 32-bit register moves*.



Decode fields		Instruction page
opc1	L	
`000`	-	VMOV (between general-purpose register and single-precision register)
`001`	-	Unallocated.
`01x`	-	Unallocated.
`10x`	-	Unallocated.
`110`	-	Unallocated.
`111`	`0`	VMSR
`111`	`1`	VMRS

Coprocessor

This section describes the encoding of the Coprocessor group. The encodings in this section are decoded from *Coprocessor and floating-point instructions* on page C2-364.



Decode fields			Decode group or instruction page
op0	op1	op2	
`0`	`00x0`	-	*Coprocessor 64-bit move*
`0`	`!= 00x0`	-	*Coprocessor load/store registers* on page C2-372
`1`	`0xxx`	`0`	CDP, CDP2
`1`	`0xxx`	`1`	*Coprocessor 32-bit move* on page C2-373

Coprocessor 64-bit move

This section describes the encoding of the Coprocessor 64-bit move instruction class. The encodings in this section are decoded from *Coprocessor*.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 4	3 0
1	1	1	o0	1	1	0	0	0	D	0	L	Rt2	Rt	coproc	opc1	CRm

Decode fields			Instruction page
o0	**D**	**L**	
`0`	`0`	`-`	Unallocated.
`0`	`1`	`0`	MCRR, MCRR2 - T1
`0`	`1`	`1`	MRRC, MRRC2 - T1
`1`	`0`	`-`	Unallocated.
`1`	`1`	`0`	MCRR, MCRR2 - T2
`1`	`1`	`1`	MRRC, MRRC2 - T2

Coprocessor load/store registers

This section describes the encoding of the Coprocessor load/store registers instruction class. The encodings in this section are decoded from *Coprocessor* on page C2-371.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	o0	1	1	0	P	U	D	W	L	Rn	CRd	coproc	imm8

Decode fields				Instruction page
o0	**P:U:W**	**L**	**Rn**	
`0`	`!= 000`	`1`	1111	LDC, LDC2 (literal) - T1
`0`	`0x1`	`0`	-	STC, STC2
`0`	`0x1`	`1`	!= 1111	LDC, LDC2 (immediate)
`0`	`010`	`0`	-	STC, STC2
`0`	`010`	`1`	!= 1111	LDC, LDC2 (immediate)
`0`	`1x0`	`0`	-	STC, STC2
`0`	`1x0`	`1`	!= 1111	LDC, LDC2 (immediate)
`0`	`1x1`	`0`	-	STC, STC2
`0`	`1x1`	`1`	!= 1111	LDC, LDC2 (immediate)
`1`	`!= 000`	`1`	1111	LDC, LDC2 (literal) - T2
`1`	`0x1`	`0`	-	STC, STC2
`1`	`0x1`	`1`	!= 1111	LDC, LDC2 (immediate)
`1`	`010`	`0`	-	STC, STC2
`1`	`010`	`1`	!= 1111	LDC, LDC2 (immediate)

Decode fields				Instruction page
o0	P:U:W	L	Rn	
1	1x0	0	-	STC, STC2
1	1x0	1	!= 1111	LDC, LDC2 (immediate)
1	1x1	0	-	STC, STC2
1	1x1	1	!= 1111	LDC, LDC2 (immediate)

Coprocessor 32-bit move

This section describes the encoding of the Coprocessor 32-bit move instruction class. The encodings in this section are decoded from *Coprocessor* on page C2-371.

15	14	13	12	11	10	9	8	7 5	4	3 0	15 12	11 8	7 5	4	3 0
1	1	1	o0	1	1	1	0	opc1	L	CRn	Rt	coproc	opc2	1	CRm

Decode fields		Instruction page
o0	L	
0	0	MCR, MCR2 - T1
0	1	MRC, MRC2 - T1
1	0	MCR, MCR2 - T2
1	1	MRC, MRC2 - T2

C2.4 Alphabetical list of instructions

Every Armv8-M instruction is listed in this section. See Chapter C1 *Instruction Set Overview* for the format of the instruction descriptions.

C2.4.1 ADC (immediate)

Add with Carry (immediate) adds an immediate value and the carry flag value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	0	1	0	S	Rn	0	imm3	Rd	imm8

ADC variant

Applies when `S == 0`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ADCS variant

Applies when `S == 1`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

C2.4.2 ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	0	1	Rm	Rdn

T1 variant

```
ADC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ADCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	1	0	1	0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

ADC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
ADC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
ADCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADCS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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 (SP plus immediate) adds an immediate value to the SP value, and writes the result to the destination register.

T1

Armv8-M

15	14	13	12	11	10 8	7 0
1	0	1	0	1	Rd	imm8

T1 variant

```
ADD{<c>}{<q>} <Rd>, SP, #<imm8>
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm8:'00', 32);
```

T2

Armv8-M

15	14	13	12	11	10	9	8	7	6 0
1	0	1	1	0	0	0	0	0	imm7

T2 variant

```
ADD{<c>}{<q>} {SP,} SP, #<imm7>
```

Decode for this encoding

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

T3

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	0	0	0	S	1	1	0	1	0	imm3	Rd	imm8

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 all variants of 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

15 14 13 12 11	10	9 8	7	6	5	4	3 2 1 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	0	0	0	0	1 1 0 1	0	imm3	Rd	imm8

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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 *Modified immediate constants* on page C1-318 for the range of values.

Operation for all encodings

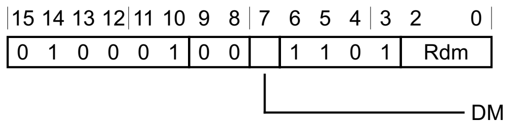
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    RSPCheck[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

C2.4.4 ADD (SP plus register)

ADD (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

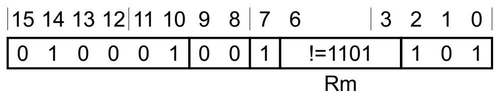
```
ADD{<c>}{<q>} {<Rdm>,} SP, <Rdm>
```

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

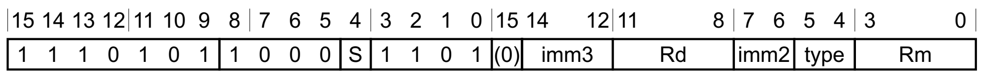
```
ADD{<c>}{<q>} {SP,} SP, <Rm>
```

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 && type == 11`.

```
ADD{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
ADDS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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 "type" field. It can have the following values:

LSL	when `type = 00`
LSR	when `type = 01`
ASR	when `type = 10`
ROR	when `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, shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        RSPCheck[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

C2.4.5 ADD (immediate)

Add (immediate) adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	1	0	imm3	Rn	Rd

T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
ADDS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

T2

Armv8-M

15	14	13	12	11	10 8	7 0
0	0	1	1	0	Rdn	imm8

T2 variant

```
ADD<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
ADD<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
ADDS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
ADDS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

T3

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	0	0	0	S	!=1101	0	imm3	Rd	imm8
												Rn				

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 all variants of 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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	0	0	0	0	!=11x1	0	imm3	Rd	imm8
												Rn				

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rdn>`	Is the general-purpose source and destination register, encoded in the "Rdn" field.
`<imm8>`	Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.
`<Rn>`	For encoding T1: is the general-purpose source register, encoded in the "Rn" field. For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD (SP plus immediate). For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see ADD (SP plus immediate). If the PC is used, see ADR.
`<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 *Modified immediate constants* on page C1-318 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)

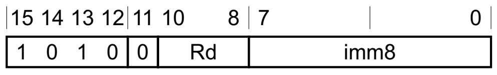
Add to PC adds an immediate value to the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

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>
```

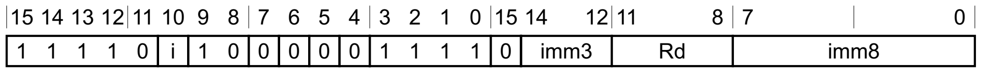
is equivalent to

```
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.

```
ADD{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is never the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<label>`	For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020. For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.
`<imm8>`	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
`<imm12>`	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

Operation for all encodings

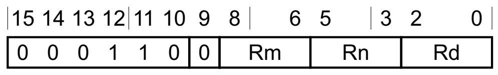
The description of ADR gives the operational pseudocode for this instruction.

C2.4.7 ADD (register)

ADD (register) 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

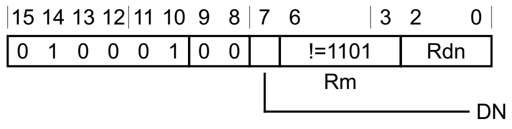
```
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
ADDS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M



T2 variant

Applies when !(DN == 1 && Rdn == 101).

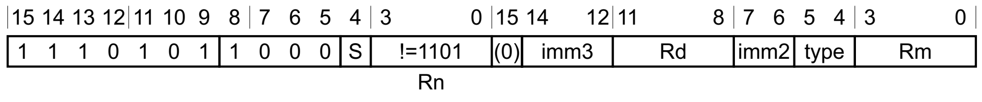
```
ADD<c>{<q>} <Rdn>, <Rm> // Preferred syntax, Inside IT block
ADD{<c>}{<q>} {<Rdn>,} <Rdn>, <Rm>
```

Decode for this encoding

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)";
d = UInt(DN:Rdn);  n = UInt(DN:Rdn);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if d == 15 && m == 15 then UNPREDICTABLE;
```

T3

Armv8-M Main Extension only



ADD, rotate right with extend variant

Applies when S == 0 && imm3 == 000 && imm2 == 00 && type == 11.

```
ADD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADD, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
ADDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ADDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (register)";
if Rn == '1101' then SEE "ADD (SP plus register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page C1-310.

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<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 ADD (SP plus register).

`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

C2.4.8 ADR

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). The pseudo-instruction is never the preferred disassembly.

T1

Armv8-M

15 14 13 12	11	10 8	7 0
1 0 1 0	0	Rd	imm8

T1 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
d = UInt(Rd);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

T2

Armv8-M Main Extension only

15 14 13 12 11	10	9 8	7	6	5	4	3 2 1 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	1	0	1	0	1 1 1 1	0	imm3	Rd	imm8

T2 variant

```
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = FALSE;
if d IN {13,15} then UNPREDICTABLE;
```

T3

Armv8-M Main Extension only

15 14 13 12 11	10	9 8	7	6	5	4	3 2 1 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	0	0	0	0	1 1 1 1	0	imm3	Rd	imm8

T3 variant

```
ADR{<c>}.W <Rd>, <label> // <Rd>, <label> can be presented in T1
ADR{<c>}{<q>} <Rd>, <label>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  imm32 = ZeroExtend(i:imm3:imm8, 32);  add = TRUE;
if d IN {13,15} then UNPREDICTABLE;
```

Alias conditions

Alias or pseudo-instruction	is preferred when
ADD (immediate, to PC)	Never
SUB (immediate, from PC)	`i:imm3:imm8 == '000000000000'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<label>`	For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020. For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    R[d] = result;
```

C2.4.9 AND (immediate)

AND (immediate) performs a bitwise AND of a register value and an immediate value, and writes the result to the destination register.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	0 0 0 0	S	Rn	0	imm3	Rd	imm8

AND variant

Applies when `S == 0`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

ANDS variant

Applies when `S == 1 && Rd != 1111`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TST (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.10 AND (register)

AND (register) performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	0	0	Rm	Rdn

T1 variant

```
AND<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ANDS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	0	0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

AND, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
AND{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

AND, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
ANDS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ANDS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of 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(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`
`ROR`	when `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)

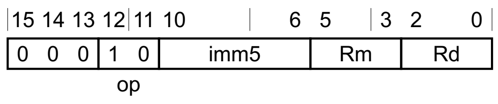
Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register.

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`

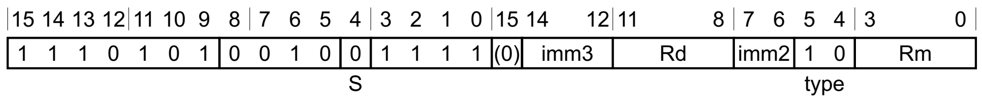
is equivalent to

`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.

`ASR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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)

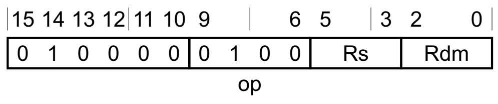
Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination registers. The variable number of bits is read from the bottom byte of a register.

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
```

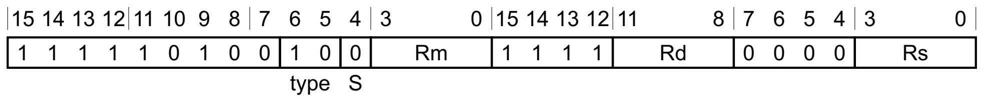
is equivalent to

```
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>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

```
ASR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

C2.4.13 ASRS (immediate)

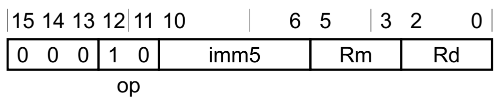
Arithmetic Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result.

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`

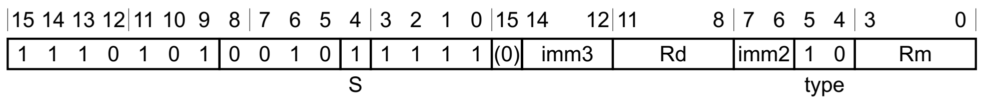
is equivalent to

`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.

`ASRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>`

is equivalent to

`MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. For encoding T3: is the shift amount, in the range 1 to 32, encoded in the "imm3:imm2" field as <imm> modulo 32.

Operation for all encodings

The description of MOV (register) gives the operational pseudocode for this instruction.

C2.4.14 ASRS (register)

Arithmetic Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

T1

Armv8-M

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

Arithmetic shift right variant

```
ASRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 2 1 0	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
1 1 1 1 1 0 1 0 0	1 0	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
	type	S					

Flag setting variant

```
ASRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

```
ASRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

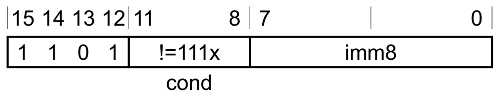
The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

C2.4.15 B

Branch causes a branch to a target address.

T1

Armv8-M



T1 variant

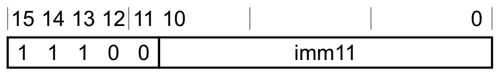
```
B<c>{<q>} <label> // Not permitted in IT block
```

Decode for this encoding

```
if cond == '1110' then SEE UDF;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

T2

Armv8-M



T2 variant

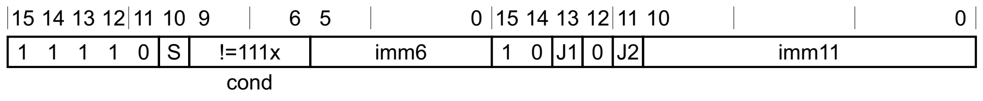
```
B{<c>}{<q>} <label> // Outside or last in IT block
```

Decode for this encoding

```
imm32 = SignExtend(imm11:'0', 32);
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

15	14	13	12	11	10	9 … 0	15	14	13	12	11	10 … 0
1	1	1	1	0	S	imm10	1	0	J1	1	J2	imm11

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);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Notes for all encodings

Related encodings: *Branches and miscellaneous control* on page C2-347.

Assembler symbols

`<c>`	For encoding T1: see *Standard assembler syntax fields* on page C1-310. Must not be `AL` or omitted. For encoding T2 and T4: see *Standard assembler syntax fields* on page C1-310. For encoding T3: see *Standard assembler syntax fields* on page C1-310. `<c>` must not be `AL` or omitted.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<label>`	For encoding T1: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –256 to 254. For encoding T2: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –2048 to 2046. For encoding T3: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –1048576 to 1048574. For encoding T4: the label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the `B` instruction to this label, then selects an encoding that sets `imm32` to that offset. Permitted offsets are even numbers in the range –16777216 to 16777214.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

C2.4.16 BFC

Bit Field Clear clears any number of adjacent bits at any position in a register, without affecting the other bits in the register.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 6	5	4 0
1	1	1	1	0	(0)	1	1	0	1	1	0	1	1	1	1	0	imm3	Rd	imm2	(0)	msb

T1 variant

```
BFC{<c>}{<q>} <Rd>, #<lsb>, #<width>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if msbit < lsbit then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = Replicate('0', msbit-lsbit+1);
        // Other bits of R[d] are unchanged
    else
        R[d] = bits(32) UNKNOWN;
```

C2.4.17 BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5	4 0
1	1	1	1	0	(0)	1	1	0	1	1	0	!=1111	0	imm3	Rd	imm2	(0)	msb
												Rn						

T1 variant

```
BFI{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if Rn == '1111' then SEE BFC;
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  msbit = UInt(msb);  lsbit = UInt(imm3:imm2);
if msbit < lsbit then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit >= lsbit then
        R[d]<msbit:lsbit> = R[n]<(msbit-lsbit):0>;
        // Other bits of R[d] are unchanged
    else
        R[d] = bits(32) UNKNOWN;
```

C2.4.18 BIC (immediate)

Bit Clear (immediate) performs a bitwise AND of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	0 0 0 1	S	Rn	0	imm3	Rd	imm8

BIC variant

Applies when `S == 0`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

BICS variant

Applies when `S == 1`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.19 BIC (register)

Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 … 3	2 … 0
0	1	0	0	0	0	1	1	1	0	Rm	Rdn

T1 variant

```
BIC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
BICS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11 … 8	7 6	5 4	3 … 0
1	1	1	0	1	0	1	0	0	0	1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

BIC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
BIC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BIC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
BICS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

BICS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
```

C2.4.20 BKPT

Breakpoint causes a DebugMonitor exception or a debug halt to occur depending on the configuration of the debug support.

Note

`BKPT` is an unconditional instruction and executes as such both inside and outside an `IT` instruction block.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	0
1	0	1	1	1	1	1	0	imm8	

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

`<q>`	See *Standard assembler syntax fields* on page C1-310. 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

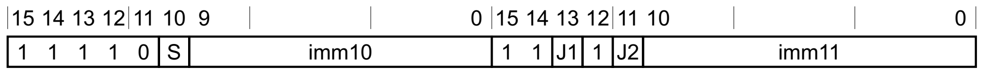
```
EncodingSpecificOperations();
BKPTInstrDebugEvent();
```

C2.4.21 BL

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

```
I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    next_instr_addr = PC;
    LR = next_instr_addr<31:1> : '1';
    BranchWritePC(PC + imm32);
```

C2.4.22 BLX, BLXNS

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.

See *Function calls from Secure state to Non-secure state* on page B3-73 for further details of register and stack changes as a result of BLXNS causing a transition from Secure to Non-secure state.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6			3	2	1	0
0	1	0	0	0	1	1	1	1	Rm				NS	(0)	(0)

BLX variant

Applies when `NS == 0`.

```
BLX{<c>}{<q>} <Rm>
```

BLXNS variant

Applies when `NS == 1`.

```
BLXNS{<c>}{<q>} <Rm>
```

Decode for all variants of 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();

    target        = R[m];
    nextInstrAddr = PC - 2;
    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;

        BLXWritePC(target, allowNonSecure);
```

CONSTRAINED UNPREDICTABLE behavior

If `!IsAligned(SP, 8)`, then one of the following behaviors must occur:

- The instruction uses the current value of the stack pointer.
- The instruction behaves as though bits[2:0] of the stack pointer are 000.

C2.4.23 BX, BXNS

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	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 all variants of 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    exc = BXWritePC(R[m], allowNonSecure);
    HandleException(exc);
```

C2.4.24 CBNZ, CBZ

Compare and Branch on Nonzero and Compare and Branch on Zero compare the value in a register with zero, and conditionally branch forward a constant value. They do not affect the condition flags.

T1

Armv8-M

15	14	13	12	11	10	9	8	7 3	2 0
1	0	1	1	op	0	i	1	imm5	Rn

CBNZ variant

Applies when `op == 1`.

```
CBNZ{<q>} <Rn>, <label>
```

CBZ variant

Applies when `op == 0`.

```
CBZ{<q>} <Rn>, <label>
```

Decode for all variants of this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(i:imm5:'0', 32);  nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
EncodingSpecificOperations();
if nonzero != IsZero(R[n]) then
    BranchWritePC(PC + imm32);
```

C2.4.25 CDP, CDP2

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

15	14	13	12	11	10	9	8	7 4	3 0	15 12	11 8	7 5	4	3 0
1	1	1	0	1	1	1	0	opc1	CRn	CRd	!=101x	opc2	0	CRm
											coproc			

T1 variant

```
CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
cp = UInt(coproc);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7 4	3 0	15 12	11 8	7 5	4	3 0
1	1	1	1	1	1	1	0	opc1	CRn	CRd	!=101x	opc2	0	CRm
											coproc			

T2 variant

```
CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
cp = UInt(coproc);
```

Notes for all encodings

See Floating-point: *Floating-point data-processing* on page C2-366.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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());
```

C2.4.26 CLREX

Clear Exclusive clears the local record of the executing PE that an address has had a request for an exclusive access.

T1

Armv8-M

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

T1 variant

```
CLREX{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

C2.4.27 CLZ

Count Leading Zeros returns the number of binary zero bits before the first binary one bit in a value.

T1

Armv8-M Main Extension only

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

T1 variant

```
CLZ{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `!Consistent(Rm)`, 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

C2.4.28 CMN (immediate)

Compare Negative (immediate) adds a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 10 9 8	7 0
1 1 1 1 0	i	0	1 0 0 0	1	Rn	0	imm3	1 1 1 1	imm8

T1 variant

```
CMN{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

C2.4.29 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M

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

T1 variant

```
CMN{<c>}{<q>} <Rn>, <Rm>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9	8 7 6 5	4	3 0	15	14 12	11 10 9 8	7 6	5 4	3 0
1 1 1 0 1 0 1	1 0 0 0	1	Rn	(0)	imm3	1 1 1 1	imm2	type	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && type == 11`.

```
CMN{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.

```
CMN{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
CMN{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`
`ROR`	when `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;
```

C2.4.30 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M

15	14	13	12	11	10 … 8	7 … 0
0	0	1	0	1	Rn	imm8

T1 variant

```
CMP{<c>}{<q>} <Rn>, #<imm8>
```

Decode for this encoding

```
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11	10	9	8	7 … 0
1	1	1	1	0	i	0	1	1	0	1	1	Rn	0	imm3	1	1	1	1	imm8

T2 variant

```
CMP{<c>}.W <Rn>, #<const> // <Rn>, <const> can be represented in T1
CMP{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rn>`	For encoding T1: is a general-purpose source register, encoded in the "Rn" field. 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 *Modified immediate constants* on page C1-318 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

C2.4.31 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M

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

T1 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> both from R0-R7
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M

15	14	13	12	11	10	9	8	7	6 3	2 0
0	1	0	0	0	1	0	1	N	Rm	Rn

T2 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // <Rn> and <Rm> not both from R0-R7
```

Decode for this encoding

```
n = UInt(N:Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11	10	9	8	7 6	5 4	3 0
1	1	1	0	1	0	1	1	1	0	1	1	Rn	(0)	imm3	1	1	1	1	imm2	type	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && type == 11`.

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page C1-310.

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<Rn>` For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the first general-purpose source register, encoded in the "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 "type" field. It can have the following values:

LSL	when `type = 00`
LSR	when `type = 01`
ASR	when `type = 10`
ROR	when `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');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

C2.4.32 CPS

Change PE State. The instruction modifies the PRIMASK and FAULTMASK special-purpose register values.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	(0)	I	F

CPSID variant

Applies when `im == 1`.

```
CPSID{<q>} <iflags>
```

CPSIE variant

Applies when `im == 0`.

```
CPSIE{<q>} <iflags>
```

Decode for all variants of 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`.

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

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:

`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.
`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.

Operation

```
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';
```

C2.4.33 DBG

Debug Hint provides a hint to debug trace support and related debug systems. See debug architecture documentation for what use (if any) is made of this instruction.

`DBG` is a NOP-compatible hint. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

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 0
1	1	1	1	0	0	1	1	1	0	1	0	(1)	(1)	(1)	(1)	1	0	(0)	0	(0)	0	0	0	1	1	1	1	option

T1 variant

```
DBG{<c>}{<q>} #<option>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// Any decoding of 'option' is specified by the debug system
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<option>`	Is a 4-bit unsigned immediate, in the range 0 to 15, encoded in the "option" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Debug(option);
```

C2.4.34 DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the `DMB` instruction are observed before any explicit memory accesses that appear in program order after the `DMB` instruction. It does not affect the ordering of any other instructions executing on the PE.

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 0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	1	option

T1 variant

```
DMB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<option>`	Specifies an optional limitation on the barrier operation. Values are:

`SY`	Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

C2.4.35 DSB

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes only when both:

- Any explicit memory access made before this instruction is complete.
- The side-effects of any SCS access that performs a context-altering operation are visible.

T1

Armv8-M

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

T1 variant

```
DSB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<option>`	Specifies an optional limitation on the barrier operation. Values are:

`SY`	Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

C2.4.36 EOR (immediate)

Exclusive OR (immediate) performs a bitwise Exclusive OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	0	1	0	0	S	Rn	0	imm3	Rd	imm8

EOR variant

Applies when `S == 0`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

EORS variant

Applies when `S == 1 && Rd != 1111`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.37 EOR (register)

Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	0	0	1	Rm	Rdn

T1 variant

```
EOR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
EORS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	1	0	0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

EOR, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
EOR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EOR, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
EORS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

EORS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of 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(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`
`ROR`	when `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
```

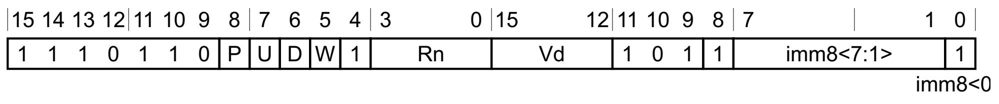
C2.4.38 FLDMDBX, FLDMIAX

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 all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
CheckDecodeFaults();
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;
if n == 15 then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `FLDMX` with the same addressing mode but loads no registers.

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.

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as `NOP`.

Notes for all encodings

Related encodings: *Floating-point load/store and 64-bit register moves* on page C2-364.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
        // 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 applylimit && (UInt(address) < UInt(limit)) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to HardFault
                excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
                HandleException(excInfo);

    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(regval) >= UInt(limit)) then
        for r = 0 to regs-1
            if single_regs then
                S[d+r]  = MemA[address,4];
                address = address+4;
            else
                word1   = MemA[address,4];  word2 = MemA[address+4,4];
                address = address+8;
                // Combine the word-aligned words in the correct order for
                // current endianness.
                D[d+r] = if BigEndian() then word1:word2 else word2:word1;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

C2.4.39 FSTMDBX, FSTMIAX

FSTMX (Decrement Before, Increment After) stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

Arm deprecates use of FSTMDBX and FSTMIAX, except for disassembly purposes, and reassembly of disassembled code.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn	Vd	1	0	1	1	imm8<7:1>	1
																			imm8<0

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 all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
CheckDecodeFaults();
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;
if n == 15 then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `FSTMX` with the same addressing mode but stores no registers.

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.

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.

Notes for all encodings

Related encodings: *Floating-point load/store and 64-bit register moves* on page C2-364.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(regval) >= UInt(limit)) 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.
                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>;
                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;
```

C2.4.40 ISB

Instruction Synchronization Barrier flushes the pipeline in the PE and is a context synchronization event. For more information, see *Instruction Synchronization Barrier* on page B5-150.

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			0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0	option			

T1 variant

```
ISB{<c>}{<q>} {<option>}
```

Decode for this encoding

```
// No additional decoding required
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<option>`	Specifies an optional limitation on the barrier operation. Values are:

`SY` Full system barrier operation, encoded as option = `0b1111`. Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as full system barrier operations, but must not be relied upon by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

C2.4.41 IT

If Then makes up to four following instructions (the IT block) conditional. The conditions for the instructions in the IT block can be the same, or some of them can be the inverse of others.

`IT` does not affect the condition code flags. Branches to any instruction in the IT block are not permitted, apart from those performed by exception returns.

16-bit instructions in the IT block, other than `CMP`, `CMN`, and `TST`, do not set the condition code flags. The `AL` condition can be specified to get this changed behavior without conditional execution.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7 4	3 0
1	0	1	1	1	1	1	1	firstcond	!=0000
									mask

T1 variant

```
IT{<x>{<y>{<z>}}}{<q>} <cond>
```

Decode for this encoding

```
if mask == '0000' then SEE "Related encodings";
if !HaveMainExt() then UNDEFINED;
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `firstcond == '1111'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes with the additional decode: `firstcond` = '1110';.

If `firstcond == '1110' && BitCount(mask) != 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.

Notes for all encodings

Related encodings: *Hints* on page C2-348.

Assembler symbols

<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:

T	`firstcond[0]`
E	`NOT 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:

T	`firstcond[0]`

`E`	`NOT 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:

`T`	`firstcond[0]`
`E`	`NOT firstcond[0]`

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<cond>` The condition for the first instruction in the IT block, encoded in the "firstcond" field. See *Conditional execution* on page C1-311 for the range of conditions available, and the encodings.

Operation

```
EncodingSpecificOperations();
ITSTATE<7:0> = firstcond:mask;
```

C2.4.42 LDA

Load-Acquire Word loads a word from memory and writes it to a register. The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
LDA{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = MemO[address, 4];
```

C2.4.43 LDAB

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

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

T1 variant

```
LDAB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 1], 32);
```

C2.4.44 LDAEX

Load-Acquire Exclusive Word loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
LDAEX{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address, 4);
    R[t] = MemO[address, 4];
```

C2.4.45 LDAEXB

Load-Acquire Exclusive Byte loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
LDAEXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);
```

C2.4.46 LDAEXH

Load-Acquire Exclusive Halfword loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
LDAEXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

C2.4.47 LDAH

Load-Acquire Halfword loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
LDAH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

C2.4.48 LDC, LDC2 (immediate)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, 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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	0	1	1	0	P	U	D	W	1	!=1111	CRd	!=101x	imm8
												Rn		coproc	

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 all variants of this encoding

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if coproc IN '101x' then SEE "Floating-point";
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');
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	1	1	1	0	P	U	D	W	1	!=1111	CRd	!=101x	imm8
												Rn		coproc	

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 all variants of this encoding

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if coproc IN '101x' then SEE "Floating-point";
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');
```

Notes for all encodings

See Floating-point: *Floating-point load/store* on page C2-365.

Assembler symbols

`L`	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 LDC, LDC2 (literal).
`<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:

-	when `U = 0`
+	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
            (limit, applylimit) = LookUpSPLim(LookUpSP());
        else
            applylimit = FALSE;

        // Memory operation only performed if limit not violated
        if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.49 LDC, LDC2 (literal)

Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11 8	7 0
1	1	1	0	1	1	0	P	U	D	W	1	1	1	1	1	CRd	!=101x	imm8
																	coproc	

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

```
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if coproc IN '101x' then SEE "Floating-point";
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` (immediate) with writeback to the PC.

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11 8	7 0
1	1	1	1	1	1	0	P	U	D	W	1	1	1	1	1	CRd	!=101x	imm8
																	coproc	

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 P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
if coproc IN '101x' 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` (immediate) with writeback to the PC.

Notes for all encodings

Floating-point: *Floating-point load/store* on page C2-365.

Assembler symbols

`L`	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

-	when `U = 0`
+	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 (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());
```

C2.4.50 LDM, LDMIA, LDMFD

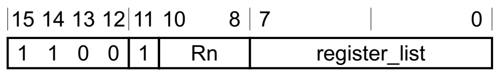
Load Multiple loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address just above the last of those locations can optionally be written back to the base register.

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). The alias is always the preferred disassembly.

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

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

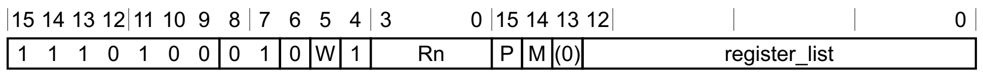
CONSTRAINED UNPREDICTABLE behavior

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

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

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.

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.

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.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

T3

Armv8-M

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	1	1	0	P	register_list

T3 variant

```
LDM{<c>}{<q>} SP!, <registers>
```

Decode for this encoding

```
n = 13; wback = TRUE;
registers = P:'0000000':register_list;
if BitCount(registers) < 1 then UNPREDICTABLE;
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

`IA`	Is an optional suffix for the Increment After form.
`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

- 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if n == 13 && wback then
        (limit, applylimit) = LookUpSPLim(LookUpSP());
        // 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 applylimit && (UInt(address) < UInt(limit)) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to HardFault
                excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
                HandleException(excInfo);
    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 || (UInt(address) >= UInt(limit))) then
            if i != n then
                R[i]       = MemA[address,4];
            else
                newBaseVal = MemA[address,4];
            address = address + 4;
    if registers<15> == '1' && (!applylimit || (UInt(address) >= UInt(limit))) 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 may raise a fault
    // Likewise if R[n] is the SP writing back may 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;
```

C2.4.51 LDMDB, LDMEA

Load Multiple Decrement Before (Load Multiple Empty Ascending) loads multiple registers from sequential memory locations using an address from a base register. The sequential memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

The registers loaded can include the PC. If they do, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12 … 0
1	1	1	0	1	0	0	1	0	0	W	1	Rn	P	M	(0)	register_list

T1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
LDMEA{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  registers = P:M:'0':register_list;  wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

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.

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.

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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: • 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.

Operation

```
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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
        doOperation         = (!applylimit || (UInt(address) >= UInt(limit)));
    else
        doOperation         = TRUE;

    for i = 0 to 15
        // Memory operation only performed if limit not violated
        if registers<i> == '1' && doOperation 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;
    else
        newBaseVal = R[n] - 4*BitCount(registers);
    // If the PC is in the register list update that now, which may raise a fault
    if registers<15> == '1' then
```

```
        LoadWritePC(newPCVal, n, newBaseVal, wback, TRUE);
    elsif wback then
        RSPCheck[n] = newBaseVal;
```

C2.4.52 LDR (immediate)

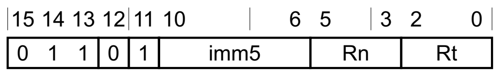
Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

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). See *Alias conditions* on page C2-464 for details of when each alias is preferred.

T1

Armv8-M



T1 variant

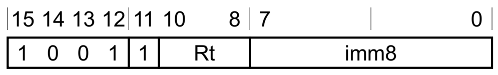
```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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

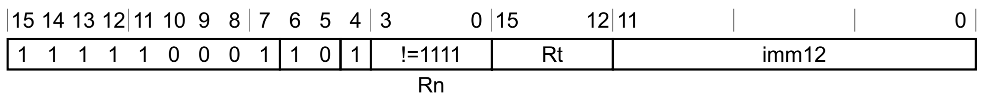
```
LDR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	1	0	1	!=1111	Rt	1	P	U	W	imm8
												Rn						

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 all variants of 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	is preferred when
POP (single register)	`Rn == '1101' && U == '1' && imm8 == '00000100'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

`<Rt>` For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.

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.

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.

For encoding T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDR (literal).

`+/-` Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`+` 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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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.

C2.4.53 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

T1

Armv8-M

15	14	13	12	11	10 8	7 0
0	1	0	0	1	Rt	imm8

T1 variant

```
LDR{<c>}{<q>} <Rt>, <label> // Normal form
```

Decode for this encoding

```
t = UInt(Rt);  imm32 = ZeroExtend(imm8:'00', 32);  add = TRUE;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11 0
1	1	1	1	1	0	0	0	U	1	0	1	1	1	1	1	Rt	imm12

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

```
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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:
	`-` when `U = 0`
	`+` when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation for all encodings

```
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.

C2.4.54 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, so bit[0] must be 1. If bit[0] is 0 the PE takes an INVSTATE UsageFault exception on the instruction at the target address.

T1

Armv8-M

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

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8 7 6	5 4	3 0
1 1 1 1 1 0 0 0 0	1 0	1	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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:

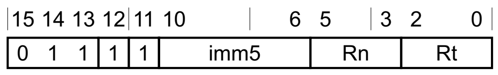
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction generates an UNALIGNED UsageFault.

C2.4.55 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M



T1 variant

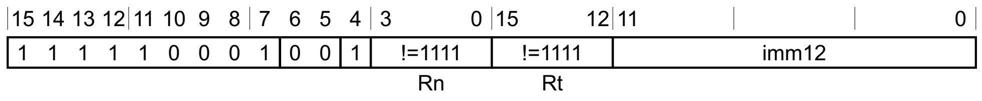
```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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

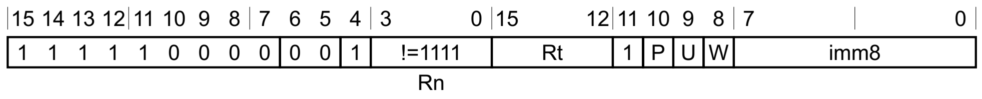
```
LDRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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 all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '1' && U == '1' && W == '0' then SEE LDRBT;
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 t == 13 || (wback && n == t) then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRB (literal).

`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	`-` when `U = 0`
	`+` when `U = 1`

`+`	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 an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. 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. 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

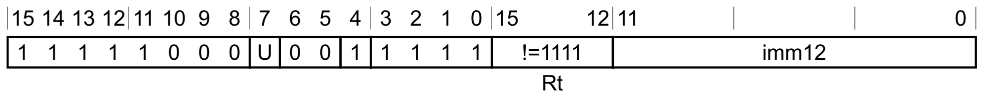
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
        R[t] = ZeroExtend(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;
```

C2.4.56 LDRB (literal)

Load Register Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register.

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

```
if Rt == '1111' then SEE "PLD (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address,1], 32);
```

C2.4.57 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M

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

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8 7 6	5 4	3 0
1 1 1 1 1 0 0 0 0	0 0	1	!=1111	!=1111	0 0 0 0 0 0	imm2	Rm
			Rn	Rt			

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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);
```

C2.4.58 LDRBT

Load Register Byte Unprivileged calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register.

When privileged software uses an `LDRBT` instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8	7 0
1 1 1 1 1 0 0 0 0	0 0	1	!=1111	Rt	1 1 1 0	imm8
			Rn			

T1 variant

```
LDRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRB (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = ZeroExtend(MemU_unpriv[address,1],32);
```

C2.4.59 LDRD (immediate)

Load Register Dual (immediate) calculates an address from a base register value and an immediate offset, loads two words from memory, and writes them to two registers. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	0	1	0	0	P	U	1	W	1	!=1111	Rt	Rt2	imm8
												Rn			

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 all variants of this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRD (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
```

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.

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.

Notes for all encodings

Related encodings: *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 LDRD (literal).
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	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

```
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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.60 LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers.

Note

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11 8	7 0
1	1	1	0	1	0	0	1	U	1	0	1	1	1	1	1	Rt	Rt2	imm8
							P			W								

T1 variant

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // Normal form
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // Alternative form
```

Decode for this encoding

```
if P == '0' && W == '0'            then SEE "Related encodings";
if P == '1' && W == '1' && U == '0' then SEE SG;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32);  add = (U == '1');
if t IN {13,15} || t2 IN {13,15} || t == t2 then UNPREDICTABLE;
if W == '1' then UNPREDICTABLE;
```

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.

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. The instruction is handled as described in *Registers* on page B3-41.

Notes for all encodings

Related encodings: *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

`<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:

-	when `U = 0`
+	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

```
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.

C2.4.61 LDREX

Load Register Exclusive calculates an address from a base register value and an immediate offset, loads a word from memory, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	0	0	0	0	1	0	1	Rn	Rt	(1)	(1)	(1)	(1)	imm8

T1 variant

```
LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    SetExclusiveMonitors(address,4);
    R[t] = MemA[address,4];
```

C2.4.62 LDREXB

Load Register Exclusive Byte derives an address from a base register value, loads a byte from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

T1

Armv8-M

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

T1 variant

```
LDREXB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

C2.4.63 LDREXH

Load Register Exclusive Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it to form a 32-bit word, writes it to a register, and:

- If the address has the Shareable memory attribute, marks the physical address as exclusive access for the executing PE in a global monitor.
- Causes the executing PE to indicate an active exclusive access in the local monitor.

T1

Armv8-M

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

T1 variant

```
LDREXH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```

C2.4.64 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M

15	14	13	12	11	10 6	5 3	2 0
1	0	0	0	1	imm5	Rn	Rt

T1 variant

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 0
1	1	1	1	1	0	0	0	1	0	1	1	!=1111	!=1111	imm12
												Rn	Rt	

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

```
if Rt == '1111' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRH (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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	0	1	1	!=1111	Rt	1	P	U	W	imm8
												Rn						

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 all variants of this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '1' && W == '0' then SEE LDRHT;
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 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.

Notes for all encodings

Related encodings: For encoding T2, see *Load/store, unsigned (positive immediate)* on page C2-353. For encoding T3, see *Load/store, unsigned (negative immediate)* on page C2-352.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. For encoding T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see LDRH (literal).
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values: `-` when `U = 0`; `+` when `U = 1`
`+`	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 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.

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.

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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.65 LDRH (literal)

Load Register Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11 0
1	1	1	1	1	0	0	0	U	0	1	1	1	1	1	1	!=1111	imm12
																Rt	

T1 variant

```
LDRH{<c>}{<q>} <Rt>, <label> // Preferred syntax
LDRH{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation

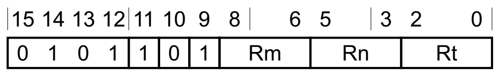
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = ZeroExtend(data, 32);
```

C2.4.66 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M



T1 variant

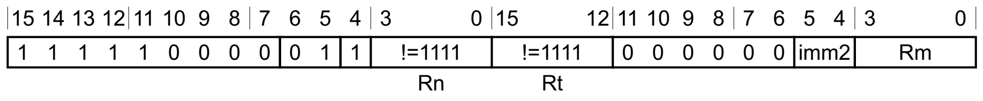
```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

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;
```

Notes for all encodings

Related encodings: *Load/store, unsigned (register offset)* on page C2-351.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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);
```

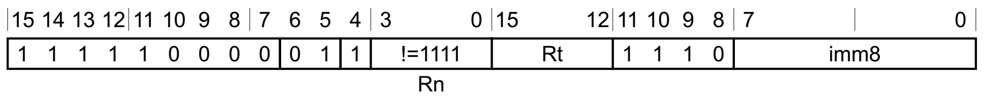
C2.4.67 LDRHT

Load Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register.

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

```
if Rn == '1111' then SEE "LDRH (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

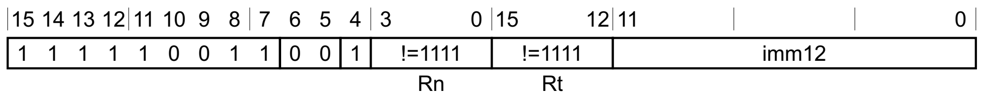
```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = ZeroExtend(data, 32);
```

C2.4.68 LDRSB (immediate)

Load Register Signed Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M Main Extension only



T1 variant

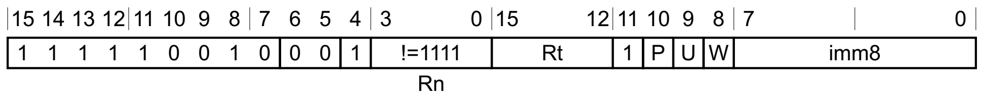
```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLI (immediate, literal)";
if Rn == '1111' then SEE "LDRSB (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;
```

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 all variants of this encoding

```
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLI (immediate, literal)";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE LDRSBT;
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 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 LDRSB (literal).
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`+`	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 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 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

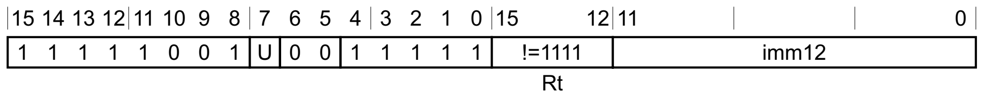
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // Determine if the stack pointer limit should be checked
    if n == 13 && wback then
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.69 LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register.

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

```
if Rt == '1111' then SEE "PLI (immediate, literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address,1], 32);
```

C2.4.70 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M

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

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5 4	3 0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111	!=1111	0	0	0	0	0	0	imm2	Rm
												Rn	Rt								

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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);
```

C2.4.71 LDRSBT

Load Register Signed Byte Unprivileged calculates an address from a base register value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register.

When privileged software uses an `LDRSBT` instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSB (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
```

C2.4.72 LDRSH (immediate)

Load Register Signed Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 0
1	1	1	1	1	0	0	1	1	0	1	1	!=1111	!=1111	imm12
												Rn	Rt	

T1 variant

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<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);  imm32 = ZeroExtend(imm12, 32);
index = TRUE;  add = TRUE;  wback = FALSE;
if t == 13 then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	1	0	0	1	1	!=1111	Rt	1	P	U	W	imm8
												Rn						

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 all variants of this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && U == '1' && W == '0' then SEE LDRSHT;
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 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.

Notes for all encodings

Related encodings: For encoding T1, see *Load/store, signed (positive immediate)* on page C2-356. For encoding T2, see *Load/store single* on page C2-349.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 LDRSH (literal).
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	`-` when `U = 0`
	`+` when `U = 1`
`+`	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 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 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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
        R[t] = SignExtend(MemU[address,2], 32);
```

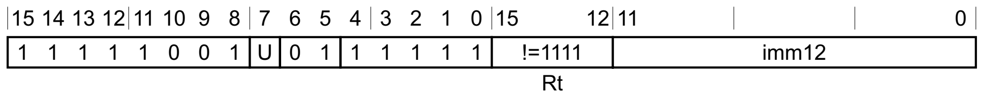
```
// If the stack pointer is being updated a fault will be raised if
// the limit is violated
if wback then RSPCheck[n] = offset_addr;
```

C2.4.73 LDRSH (literal)

Load Register Signed Halfword (literal) calculates an address from the PC value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register.

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

```
if Rt == '1111' then SEE "Related encodings";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
if t == 13 then UNPREDICTABLE;
```

Notes for all encodings

Related encodings: *Load, signed (literal)* on page C2-357.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose register to be transferred, encoded in the "Rt" field.
`<label>`	The label of the literal data item that is to be loaded into `<Rt>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. Permitted values of the offset are -4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`, encoded as U == 1. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`, encoded as U == 0.
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation

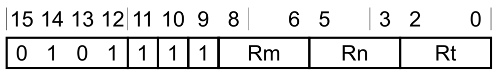
```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    data = MemU[address,2];
    R[t] = SignExtend(data, 32);
```

C2.4.74 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M



T1 variant

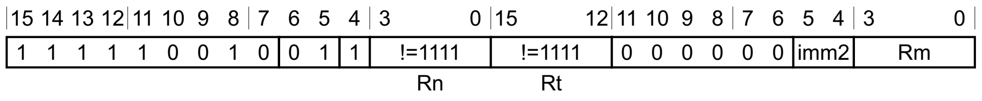
```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

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;
```

Notes for all encodings

Related encodings: *Load/store, signed (register offset)* on page C2-354.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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);
```

C2.4.75 LDRSHT

Load Register Signed Halfword Unprivileged calculates an address from a base register value and an immediate offset, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register.

When privileged software uses an `LDRSHT` instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15 14 13 12	11 10 9 8	7 6 5	4	3 0	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0 1 0	0 1	1	!=1111	Rt	1 1 1 0	imm8
				Rn			

T1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,2];
    R[t] = SignExtend(data, 32);
```

C2.4.76 LDRT

Load Register Unprivileged calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register.

When privileged software uses an `LDRT` instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	1	0	1	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
LDRT{<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);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = MemU_unpriv[address,4];
    R[t] = data;
```

C2.4.77 LSL (immediate)

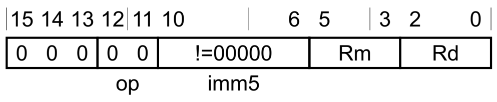
Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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
```

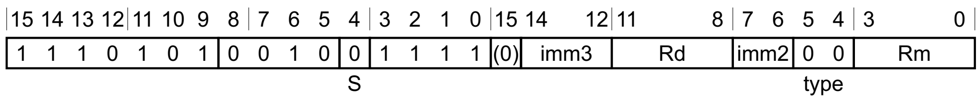
is equivalent to

```
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.

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, #<imm>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.
	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.

C2.4.78 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

T1

Armv8-M Main Extension only

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

Logical shift left variant

```
LSL<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	0	0	0	0	0	Rm	1	1	1	1	Rd	0	0	0	0	Rs
									type		S											

Non flag setting variant

```
LSL<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>, LSL <Rs>
```

and is always the preferred disassembly.

```
LSL{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

`<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.

C2.4.79 LSLS (immediate)

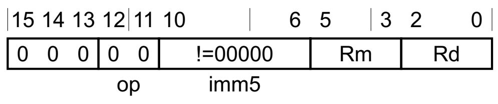
Logical Shift Left, Setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

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`

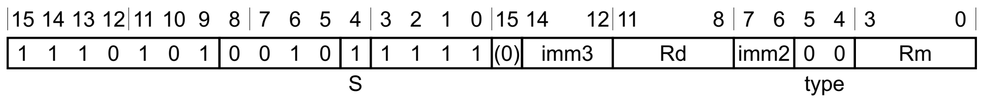
is equivalent to

`MOVS{<q>} <Rd>, <Rm>, LSL #<imm>`

and is the preferred disassembly when `!InITBlock()`.

T3

Armv8-M Main Extension only


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 15 14 12 11 8 7 6 5 4 3 0
1 1 1 0 1 0 1 | 0 0 1 0 | 1 | 1 1 1 1 | (0) | imm3 | Rd | imm2 | 0 0 | Rm
S
type

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.

`LSLS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>`

is equivalent to

`MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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.

C2.4.80 LSLS (register)

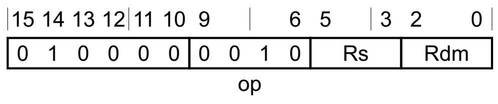
Logical Shift Left, Setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

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
```

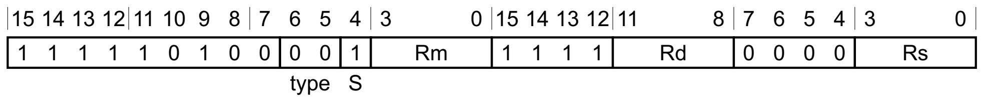
is equivalent to

```
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>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

```
LSLS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.81 LSR (immediate)

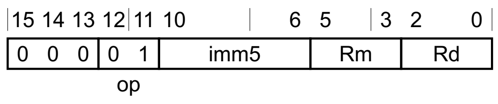
Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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`

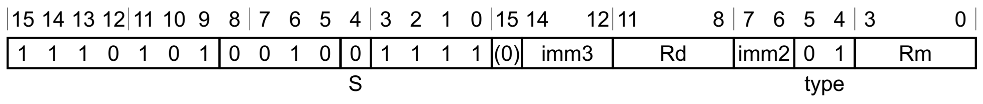
is equivalent to

`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.

`LSR{<c>}{<q>} {<Rd>,} <Rm>, #<imm>`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.
	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.

C2.4.82 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

T1

Armv8-M Main Extension only

15 14 13 12 11 10	9 … 6	5 … 3	2 … 0
0 1 0 0 0 0	0 0 1 1	Rs	Rdm
	op		

Logical shift right variant

```
LSR<c>{<q>} {<Rdm>,} <Rdm>, <Rs> // Inside IT block
```

is equivalent to

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `InITBlock()`.

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 … 0	15 14 13 12	11 … 8	7 6 5 4	3 … 0
1 1 1 1 1 0 1 0 0	0 1	0	Rm	1 1 1 1	Rd	0 0 0 0	Rs
	type	S					

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.

```
LSR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.

`<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.

C2.4.83 LSRS (immediate)

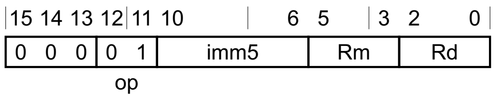
Logical Shift Right, Setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result.

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`

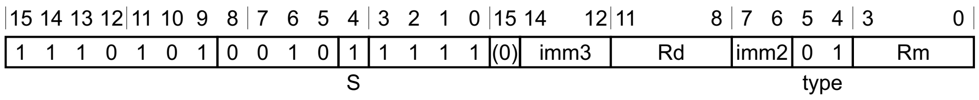
is equivalent to

`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.

`LSRS{<c>}{<q>} {<Rd>,} <Rm>, #<imm>`

is equivalent to

`MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<imm>`

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.
	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.

C2.4.84 LSRS (register)

Logical Shift Right, Setting flags (register) shifts a register value right by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

T1

Armv8-M

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

Logical shift right variant

```
LSRS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 … 0	15 14 13 12	11 … 8	7 6 5 4	3 … 0
1 1 1 1 1 0 1 0 0	0 1	1	Rm	1 1 1 1	Rd	0 0 0 0	Rs
	type	S					

Flag setting variant

```
LSRS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

```
LSRS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.85 MCR, MCR2

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

15 14 13	12	11 10 9 8	7 5	4	3 0	15 12	11 8	7 5	4	3 0
1 1 1	0	1 1 1 0	opc1	0	CRn	Rt	!=101x	opc2	1	CRm
							coproc			

T1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

15 14 13	12	11 10 9 8	7 5	4	3 0	15 12	11 8	7 5	4	3 0
1 1 1	1	1 1 1 0	opc1	0	CRn	Rt	!=101x	opc2	1	CRm
							coproc			

T2 variant

```
MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
if coproc IN '101x' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
if t == 15 || t == 13 then UNPREDICTABLE;
```

Notes for all encodings

See Floating-point: *Floating-point 32-bit move* on page C2-370.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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());
```

C2.4.86 MCRR, MCRR2

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

15 14 13	12	11 10 9 8 7	6	5	4	3 0	15 12	11 8	7 4	3 0
1 1 1	0	1 1 0 0 0	1	0	0	Rt2	Rt	!=101x	opc1	CRm
								coproc		

T1 variant

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

15 14 13	12	11 10 9 8 7	6	5	4	3 0	15 12	11 8	7 4	3 0
1 1 1	1	1 1 0 0 0	1	0	0	Rt2	Rt	!=101x	opc1	CRm
								coproc		

T2 variant

```
MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
if coproc IN '101x' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

Notes for all encodings

See Floating-point: *Floating-point 64-bit move* on page C2-365.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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());
```

C2.4.87 MLA

Multiply Accumulate multiplies two register values, and adds a third register value. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	0	0	0	0	Rn	!=1111	Rd	0	0	0	0	Rm
													Ra						

T1 variant

```
MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then SEE MUL;
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  setflags = FALSE;
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result<31:0>);
        // APSR.C unchanged
        // APSR.V unchanged
```

C2.4.88 MLS

Multiply and Subtract multiplies two register values, and subtracts the least significant 32 bits of the result from a third register value. These 32 bits do not depend on whether signed or unsigned calculations are performed. The result is written to the destination register.

T1

Armv8-M Main Extension only

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

T1 variant

```
MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]);  // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]);  // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]);  // addend   = UInt(R[a]) produces the same final results
    result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

C2.4.89 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

T1

Armv8-M

15	14	13	12	11	10 8	7 0
0	0	1	0	0	Rd	imm8

T1 variant

```
MOV<c>{<q>} <Rd>, #<imm8> // Inside IT block
MOVS{<q>} <Rd>, #<imm8> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3	Rd	imm8

MOV variant

Applies when `S == 0`.

```
MOV<c>.W <Rd>, #<const> // Inside IT block, and <Rd>, <const> can be represented in T1
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS variant

Applies when `S == 1`.

```
MOVS.W <Rd>, #<const> // Outside IT block, and <Rd>, <const> can be represented in T1
MOVS{<c>}{<q>} <Rd>, #<const>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

T3

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	0	1	0	0	imm4	0	imm3	Rd	imm8

T3 variant

```
MOV{<c>}{<q>} <Rd>, #<imm16> // <imm16> cannot be represented in T1 or T2
MOVW{<c>}{<q>} <Rd>, #<imm16> // <imm16> can be represented in T1 or T2
```

Decode for this encoding

```
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);  carry = bit UNKNOWN;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 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
```

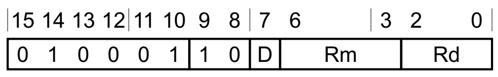
C2.4.90 MOV (register)

Move (register) copies a value from a register to the destination register. It can optionally update the condition flags based on the value.

This instruction is used by the aliases ASRS (immediate), ASR (immediate), LSLS (immediate), LSL (immediate), LSRS (immediate), LSR (immediate), RORS (immediate), ROR (immediate), RRXS, and RRX. See *Alias conditions* on page C2-530 for details of when each alias is preferred.

T1

Armv8-M



T1 variant

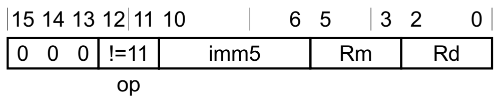
```
MOV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
if HaveMainExt() then
    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

```
if op == '11' then SEE "Related encodings";
d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = DecodeImmShift(op, imm5);
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

15 14 13 12 11 10 9	8 7 6 5	4	3 2 1 0	15	14 12	11 8	7 6	5 4	3 0
1 1 1 0 1 0 1	0 0 1 0	S	1 1 1 1	(0)	imm3	Rd	imm2	type	Rm

MOV, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOV, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if !setflags && (imm3:imm2:type == '0000000') then
    if (d == 15 || m == 15 || (d == 13 && m == 13)) then UNPREDICTABLE;
else
    if (d IN {13,15} || m IN {13,15}) then UNPREDICTABLE;
```

Notes for all encodings

Related encodings: In encoding T2, for `op` == 11, see *Add, subtract (three low registers)* on page C2-326 and *Add, subtract (two low registers and immediate)* on page C2-326.

Alias conditions

Alias	of variant	is preferred when
ASRS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && type == '10'`
ASRS (immediate)	T2	`op == '10' && !InITBlock()`
ASR (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && type == '10'`
ASR (immediate)	T2	`op == '10' && InITBlock()`

Alias	of variant	is preferred when
LSLS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && type == '00'`
LSLS (immediate)	T2	`op == '00' && imm5 != '00000' && !InITBlock()`
LSL (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && type == '00'`
LSL (immediate)	T2	`op == '00' && imm5 != '00000' && InITBlock()`
LSRS (immediate)	T3 (MOVS, shift or rotate by value)	`S == '1' && type == '01'`
LSRS (immediate)	T2	`op == '01' && !InITBlock()`
LSR (immediate)	T3 (MOV, shift or rotate by value)	`S == '0' && type == '01'`
LSR (immediate)	T2	`op == '01' && InITBlock()`
RORS (immediate)	-	`S == '1' && imm3:Rd:imm2 != '000xxxx00' && type == '11'`
ROR (immediate)	-	`S == '0' && imm3:Rd:imm2 != '000xxxx00' && type == '11'`
RRXS	-	`S == '1' && imm3 == '000' && imm2 == '00' && type == '11'`
RRX	-	`S == '0' && imm3 == '000' && imm2 == '00' && type == '11'`

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page C1-310.

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<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 `op = 00`
`LSR`	when `op = 01`
`ASR`	when `op = 10`

For encoding T3: is the type of shift to be applied to the source register, encoded in the "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`
`ROR`	when `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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        RSPCheck[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

C2.4.91 MOV, MOVS (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), and ROR (register). See *Alias conditions* on page C2-534 for details of when each alias is preferred.

T1

Armv8-M

15 14 13 12 11 10	9 6	5 3	2 0
0 1 0 0 0 0	0 x x x	Rs	Rdm
	op		

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 all variants of this encoding

```
if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
d = UInt(Rdm);  m = UInt(Rdm);  s = UInt(Rs);
setflags = !InITBlock();  shift_t = DecodeRegShift(op<2>:op<0>);
```

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 14 13 12	11 8	7 6 5 4	3 0
1 1 1 1 1 0 1 0 0	type	S	Rm	1 1 1 1	Rd	0 0 0 0	Rs

Flag setting variant

Applies when `S == 1`.

```
MOVS.W <Rd>, <Rm>, <type> <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
MOVS{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

Non flag setting variant

Applies when `S == 0`.

```
MOV<c>.W <Rd>, <Rm>, <type> <Rs> // Inside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in
T1
MOV{<c>}{<q>} <Rd>, <Rm>, <type> <Rs>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  s = UInt(Rs);
setflags = (S == '1');  shift_t = DecodeRegShift(type);
if d IN {13,15} || m IN {13,15} || s IN {13,15} then UNPREDICTABLE;
```

Notes for all encodings

Related encodings: In encoding T1, for an `op` field value that is not listed, see *Data-processing (two low registers)* on page C2-327.

Alias conditions

Alias	of variant	is preferred when
ASRS (register)	T1 (arithmetic shift right)	`op == '0100' && !InITBlock()`
ASRS (register)	T2 (flag setting)	`type == '10' && S == '1'`
ASR (register)	T1 (arithmetic shift right)	`op == '0100' && InITBlock()`
ASR (register)	T2 (non flag setting)	`type == '10' && S == '0'`
LSLS (register)	T1 (logical shift left)	`op == '0010' && !InITBlock()`
LSLS (register)	T2 (flag setting)	`type == '00' && S == '1'`
LSL (register)	T1 (logical shift left)	`op == '0010' && InITBlock()`
LSL (register)	T2 (non flag setting)	`type == '00' && S == '0'`
LSRS (register)	T1 (logical shift right)	`op == '0011' && !InITBlock()`
LSRS (register)	T2 (flag setting)	`type == '01' && S == '1'`
LSR (register)	T1 (logical shift right)	`op == '0011' && InITBlock()`
LSR (register)	T2 (non flag setting)	`type == '01' && S == '0'`
RORS (register)	T1 (rotate right)	`op == '0111' && !InITBlock()`
RORS (register)	T2 (flag setting)	`type == '11' && S == '1'`
ROR (register)	T1 (rotate right)	`op == '0111' && InITBlock()`
ROR (register)	T2 (non flag setting)	`type == '11' && S == '0'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

`<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.
`<type>`	Is the type of shift to be applied to the second source register, encoded in the "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
```

C2.4.92 MOVT

Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	1	1	0	0	imm4	0	imm3	Rd	imm8

T1 variant

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

Decode for this encoding

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

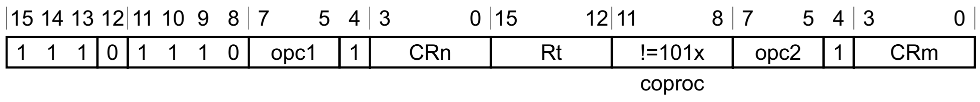
```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<31:16> = imm16;
    // R[d]<15:0> unchanged
```

C2.4.93 MRC, MRC2

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

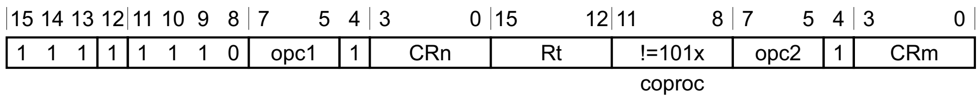
```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
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

```
if coproc IN '101x' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  cp = UInt(coproc);
if t == 13 then UNPREDICTABLE;
```

Notes for all encodings

Floating-point: *Floating-point 32-bit move* on page C2-370.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 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.
```

C2.4.94 MRRC, MRRC2

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 4	3 0
1	1	1	0	1	1	0	0	0	1	0	1	Rt2	Rt	!=101x	opc1	CRm
														coproc		

T1 variant

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
if coproc IN '101x' then SEE "Floating-point";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The value in the destination register is UNKNOWN.

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 4	3 0
1	1	1	1	1	1	0	0	0	1	0	1	Rt2	Rt	!=101x	opc1	CRm
														coproc		

T2 variant

```
MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
if coproc IN '101x' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
```

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.

Notes for all encodings

Floating-point: *Floating-point 64-bit move* on page C2-365.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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
        (R[t2], R[t]) = Coproc_GetTwoWords(cp, ThisInstr());
```

C2.4.95 MRS

Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose register.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7 0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd	SYSm

T1 variant

```
MRS{<c>}{<q>} <Rd>, <spec_reg>
```

Decode for this encoding

```
d = UInt(Rd);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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 HaveFPExt() && IsSecure() then
                R[d]<3:0> = CONTROL<3:0>;
            elsif HaveFPExt() 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 HaveFPExt() 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.

C2.4.96 MSR (register)

Move to Special register from Register moves the value of a general-purpose register to the specified special-purpose register.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 10	9	8	7 0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn	1	0	(0)	0	mask	(0)	(0)	SYSm

T1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

Decode for this encoding

```
n = UInt(Rn);
if HaveMainExt() then
    if mask == '00' || (mask != '10' && !(UInt(SYSm) IN {0..3})) then UNPREDICTABLE;
else
    if mask != '10' then UNPREDICTABLE;
if n IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If the combination of SYSm and mask are 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.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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
                        _R[RNameSP_Main_Secure] = R[n]<31:2>:'00';
                    else
                        _R[RNameSP_Main_NonSecure] = R[n]<31:2>:'00';
                when '001'
                    // MSR not subject to SP limit, write directly to register.
                    if IsSecure() then
                        _R[RNameSP_Process_Secure] = R[n]<31:2>:'00';
                    else
                        _R[RNameSP_Process_NonSecure] = R[n]<31:2>:'00';
                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.
                    _R[RNameSP_Main_NonSecure] = R[n]<31:2>:'00';
                when '001'
                    // MSR not subject to SP limit, write directly to register.
                    _R[RNameSP_Process_NonSecure] = R[n]<31:2>:'00';
                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 HaveFPExt() && (IsSecure() || NSACR.CP10 == '1') then
                        CONTROL.FPCA = R[n]<2>;
                if HaveFPExt() && 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 HaveFPExt() 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, R[n]);
                otherwise
                    UNPREDICTABLE;
    otherwise
        UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If SYSm is not a 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.

C2.4.97 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

It can optionally update the condition flags based on the result. In the T32 instruction set, this option is limited to only a few forms of the instruction. Use of this option adversely affects performance on many implementations.

T1

Armv8-M

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

T1 variant

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // Inside IT block
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdm);  n = UInt(Rn);  m = UInt(Rdm);  setflags = !InITBlock();
```

T2

Armv8-M Main Extension only

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

T2 variant

```
MUL<c>.W <Rd>, <Rn>{, <Rm>} // Inside IT block, and <Rd>, <Rn>, <Rm> can be represented in T1
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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
```

C2.4.98 MVN (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register. It can optionally update the condition flags based on the value.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	0	0	1	1	S	1	1	1	1	0	imm3	Rd	imm8

MVN variant

Applies when `S == 0`.

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS variant

Applies when `S == 1`.

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.99 MVN (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

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

T1 variant

```
MVN<c>{<q>} <Rd>, <Rm> // Inside IT block
MVNS{<q>} <Rd>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3	Rd	imm2	type	Rm

MVN, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
```

C2.4.100 NOP

No Operation does nothing.

The architecture makes no guarantees about any timing effects of including a NOP instruction.

This is a NOP-compatible hint. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

T1

Armv8-M

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

T1 variant

```
NOP{<c>}{<q>}
```

Decode for this encoding

```
// 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	0

T2 variant

```
NOP{<c>}.W
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    // Do nothing
```

C2.4.101 ORN (immediate)

Logical OR NOT (immediate) performs a bitwise (inclusive) OR of a register value and the complement of an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	0	0	1	1	S	!=1111	0	imm3	Rd	imm8
												Rn				

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 all variants of this encoding

```
if Rn == '1111' then SEE "MVN (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.102 ORN (register)

Logical OR NOT (register) performs a bitwise (inclusive) OR of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	1	1	S	!=1111	(0)	imm3	Rd	imm2	type	Rm
												Rn						

ORN, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORN, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORN{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

ORNS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && type == 11`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORNS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
ORNS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "MVN (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when `type = 00`
LSR	when `type = 01`

`ASR`	when `type = 10`
`ROR`	when `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

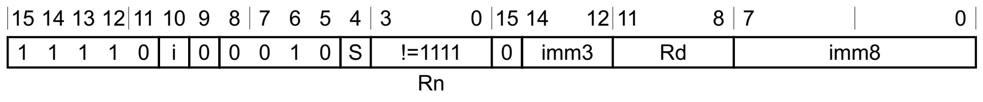
```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.103 ORR (immediate)

Logical OR (immediate) performs a bitwise (inclusive) OR of a register value and an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

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 all variants of this encoding

```
if Rn == '1111' then SEE "MOV (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} || n == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] OR imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

C2.4.104 ORR (register)

Logical OR (register) performs a bitwise (inclusive) OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	1	1	0	0	Rm	Rdn

T1 variant

```
ORR<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
ORRS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	1	0	S	!=1111	(0)	imm3	Rd	imm2	type	Rm
												Rn						

ORR, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
ORR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORR, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
ORRS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

ORRS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
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(type, imm3:imm2);
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
```

C2.4.105 PKHBT, PKHTB

Pack Halfword combines one halfword of its first operand with the other halfword of its shifted second operand.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5	4	3 0
1	1	1	0	1	0	1	0	1	1	0	0	Rn	(0)	imm3	Rd	imm2	tb	0	Rm
											S							T	

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 all variants of this encoding

```
if S == '1' || T == '1' then UNDEFINED;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`omitted`	No shift, encoded as `0b00000`.
`1-31`	Left shift by specified number of bits, encoded as a binary number.

For `PKHTB`, it is one of:

`omitted`	Instruction is a pseudo-instruction and is assembled as though `PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn>` had been written.
`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.

Note

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

```
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;
```

C2.4.106 PLD (literal)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

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	0
1	1	1	1	1	0	0	0	U	0	0	1	1	1	1	1	1	1	1	1	imm12	

T1 variant

```
PLD{<c>}{<q>} <label> // Preferred syntax
PLD{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<label>`	The label of the literal data item that is likely to be accessed in the near future. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the instruction to this label. The offset must be in the range –4095 to 4095. If the offset is zero or positive, `imm32` is equal to the offset and `add` == `TRUE`. If the offset is negative, `imm32` is equal to minus the offset and `add` == `FALSE`.
`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:

-	when `U = 0`
+	when `U = 1`

`<imm>`	Is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint_PreloadData(address);
```

C2.4.107 PLD (register)

Preload Data is a memory hint instruction that can signal the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache.

Register-offset

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11	10	9	8	7	6	5 4	3 0
1	1	1	1	1	0	0	0	0	0	W	1	!=1111	1	1	1	1	0	0	0	0	0	0	imm2	Rm
												Rn												

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 all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(shift));
if m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadData(address);
```

C2.4.108 PLD, PLDW (immediate)

Preload Data signals the memory system that data memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the data cache. 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. For more information, see *Behavior of Preload Data (PLD) and Preload Instruction (PLI) instructions with caches* on page B5-179.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 0
1	1	1	1	1	0	0	0	1	0	W	1	!=1111	1	1	1	1	imm12
												Rn					

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 all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;  is_pldw = (W == '1');
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	0	W	1	!=1111	1	1	1	1	1	1	0	0	imm8
												Rn									

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 all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;  is_pldw = (W == '1');
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see PLD (literal).
`+`	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. 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();
    address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

C2.4.109 PLI (immediate, literal)

Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 0
1	1	1	1	1	0	0	1	1	0	0	1	!=1111	1	1	1	1	imm12
												Rn					

T1 variant

```
PLI{<c>}{<q>} [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = ZeroExtend(imm12, 32);  add = TRUE;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11	10	9	8	7 0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111	1	1	1	1	1	1	0	0	imm8
												Rn									

T2 variant

```
PLI{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  imm32 = ZeroExtend(imm8, 32);  add = FALSE;
```

T3

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 0
1	1	1	1	1	0	0	1	U	0	0	1	1	1	1	1	1	1	1	1	imm12

T3 variant

```
PLI{<c>}{<q>} <label> // Preferred syntax
PLI{<c>}{<q>} [PC, #{+/-}<imm>] // Alternative syntax
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = 15;  imm32 = ZeroExtend(imm12, 32);  add = (U == '1');
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:
	`-` when `U = 0`
	`+` 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.
	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.
	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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

C2.4.110 PLI (register)

Preload Instruction is a memory hint instruction that can signal the memory system that instruction memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the instruction cache.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11	10	9	8	7	6	5 4	3 0
1	1	1	1	1	0	0	1	0	0	0	1	!=1111	1	1	1	1	0	0	0	0	0	0	imm2	Rm
												Rn												

T1 variant

```
PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);  add = TRUE;
(shift_t, shift_n) = (SRType_LSL, UInt(imm2));
if m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint_PreloadInstr(address);
```

C2.4.111 POP (multiple registers)

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.

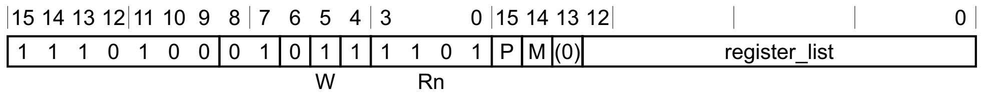
If the registers loaded include the PC, the word loaded for the PC is treated as a branch address or an exception return value. Bit[0] complies with the Arm architecture interworking rules for switching between the A32 and T32 instruction sets. However, Armv8-M only supports the T32 instruction set, 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>`

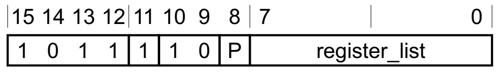
is equivalent to

`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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.112 POP (single register)

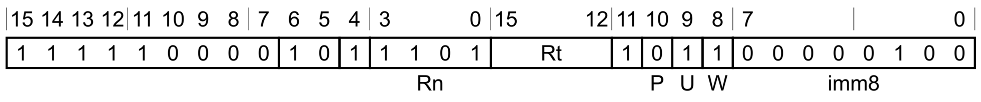
Pop single register from stack loads a single general-purpose register from the stack, loading from the address in SP, and updates SP to point above the loaded data.

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>} <single_register_list>
```

is equivalent to

```
LDR{<c>}{<q>} <Rt>, [SP], #4
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<single_register_list>`	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

The description of LDR (immediate) gives the operational pseudocode for this instruction.

C2.4.113 PUSH (multiple registers)

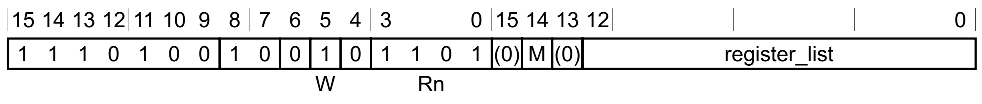
Push multiple registers to stack stores multiple general-purpose registers to the stack, storing to consecutive memory locations ending below the address in SP, and updates SP to point to the start of the stored data.

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>`

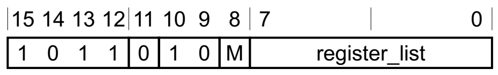
is equivalent to

`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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<registers>`	For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address. The registers in the list must be in the range R0-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The registers in the list must be in the range R0-R7, encoded in the "register_list" field, and can optionally include the LR. If the LR is in the list, the "M" field is set to 1, otherwise this field defaults to 0.

Operation for all encodings

The description of STMDB, STMFD gives the operational pseudocode for this instruction.

C2.4.114 PUSH (single register)

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11	10	9	8	7 0
1 1 1 1 1 0 0 0 0	1 0	0	1 1 0 1	Rt	1	1	0	1	0 0 0 0 0 1 0 0
			Rn			P	U	W	imm8

Pre-indexed variant

```
PUSH{<c>}{<q>} <single_register_list> // Standard syntax
```

is equivalent to

```
STR{<c>}{<q>} <Rt>, [SP, #-4]!
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<single_register_list>`	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

The description of STR (immediate) gives the operational pseudocode for this instruction.

C2.4.115 QADD

Saturating Add adds two register values, saturates the result to the 32-bit signed integer range -2^{31} to $2^{31}-1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

C2.4.116 QADD16

Saturating Add 16 performs two 16-bit integer additions, saturates the results to the 16-bit signed integer range -2^{15} to $2^{15}-1$, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    bits(32) result;
    result<15:0>  = SignedSat(sum1, 16);
    result<31:16> = SignedSat(sum2, 16);
    R[d] = result;
```

C2.4.117 QADD8

Saturating Add 8 performs four 8-bit integer additions, saturates the results to the 8-bit signed integer range -2^7 to 2^7-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    bits(32) result;
    result<7:0>   = SignedSat(sum1, 8);
    result<15:8>  = SignedSat(sum2, 8);
    result<23:16> = SignedSat(sum3, 8);
    result<31:24> = SignedSat(sum4, 8);
    R[d] = result;
```

C2.4.118 QASX

Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, saturates the results to the 16-bit signed integer range -2^{15} to $2^{15}-1$, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    bits(32) result;
    result<15:0>  = SignedSat(diff, 16);
    result<31:16> = SignedSat(sum,  16);
    R[d] = result;
```

C2.4.119 QDADD

Saturating Double and Add adds a doubled register value to another register value, and writes the result to the destination register. Both the doubling and the addition have their results saturated to the 32-bit signed integer range -2^{31} to $2^{31}-1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QDADD{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

C2.4.120 QDSUB

Saturating Double and Subtract subtracts a doubled register value from another register value, and writes the result to the destination register. Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range -2^{31} to $2^{31}-1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    (R[d], sat2)  = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        APSR.Q = '1';
```

C2.4.121 QSAX

Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, saturates the results to the 16-bit signed integer range -2^{15} to $2^{15}-1$, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    bits(32) result;
    result<15:0>  = SignedSat(sum,  16);
    result<31:16> = SignedSat(diff, 16);
    R[d] = result;
```

C2.4.122 QSUB

Saturating Subtract subtracts one register value from another register value, saturates the result to the 32-bit signed integer range -2^{31} to $2^{31}-1$, and writes the result to the destination register. If saturation occurs, it sets the Q flag in the APSR.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        APSR.Q = '1';
```

C2.4.123 QSUB16

Saturating Subtract 16 performs two 16-bit integer subtractions, saturates the results to the 16-bit signed integer range -2^{15} to $2^{15}-1$, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    bits(32) result;
    result<15:0>  = SignedSat(diff1, 16);
    result<31:16> = SignedSat(diff2, 16);
    R[d] = result;
```

C2.4.124 QSUB8

Saturating Subtract 8 performs four 8-bit integer subtractions, saturates the results to the 8-bit signed integer range -2^7 to 2^7-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
QSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0>   = SignedSat(diff1, 8);
    R[d]<15:8>  = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(diff4, 8);
```

C2.4.125 RBIT

Reverse Bits reverses the bit order in a 32-bit register.

T1

Armv8-M Main Extension only

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

T1 variant

```
RBIT{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `!Consistent(Rm)`, 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;
```

C2.4.126 REV

Byte-Reverse Word reverses the byte order in a 32-bit register.

T1

Armv8-M

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

T1 variant

```
REV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

Armv8-M Main Extension only

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

T2 variant

```
REV{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `!Consistent(Rm)`, 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	For encoding T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

Operation for all encodings

```
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;
```

C2.4.127 REV16

Byte-Reverse Packed Halfword reverses the byte order in each 16-bit halfword of a 32-bit register.

T1

Armv8-M

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

T1 variant

```
REV16{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

Armv8-M Main Extension only

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

T2 variant

```
REV16{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REV16{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `!Consistent(Rm)`, 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	For encoding T1: is the general-purpose source register, encoded in the "Rm" field.

For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

Operation for all encodings

```
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;
```

C2.4.128 REVSH

Byte-Reverse Signed Halfword reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32 bits.

T1

Armv8-M

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

T1 variant

```
REVSH{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);
```

T2

Armv8-M Main Extension only

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

T2 variant

```
REVSH{<c>}.W <Rd>, <Rm> // <Rd>, <Rm> can be represented in T1
REVSH{<c>}{<q>} <Rd>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
if !Consistent(Rm) then UNPREDICTABLE;
d = UInt(Rd);  m = UInt(Rm);
if d IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `!Consistent(Rm)`, 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.

`<Rm>`	For encoding T1: is the general-purpose source register, encoded in the "Rm" field. For encoding T2: is the general-purpose source register, encoded in the "Rm" field. Its number must be encoded twice.

Operation for all encodings

```
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;
```

C2.4.129 ROR (immediate)

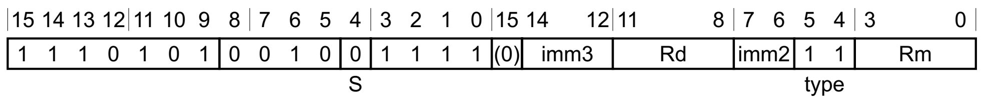
Rotate Right (immediate) rotates a register value by a constant number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

The description of MOV (register) gives the operational pseudocode for this instruction.

C2.4.130 ROR (register)

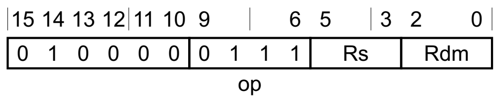
Rotate Right (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register.

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
```

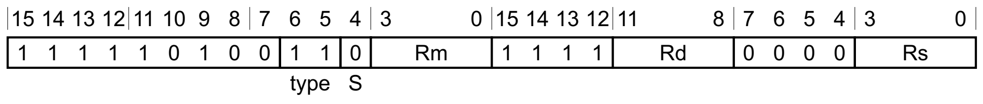
is equivalent to

```
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>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

```
ROR{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOV{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.131 RORS (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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	(0)	imm3	Rd	imm2	1 1	Rm
											S									type	

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

The description of MOV (register) gives the operational pseudocode for this instruction.

C2.4.132 RORS (register)

Rotate Right, Setting flags (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register.

This instruction is an alias of the MOV, MOVS (register-shifted register) instruction. This means that:

- The encodings in this description are named to match the encodings of MOV, MOVS (register-shifted register).
- The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

T1

Armv8-M

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

Rotate right variant

```
RORS{<q>} {<Rdm>,} <Rdm>, <Rs> // Outside IT block
```

is equivalent to

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs>
```

and is the preferred disassembly when `!InITBlock()`.

T2

Armv8-M Main Extension only

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

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

Flag setting variant

```
RORS.W {<Rd>,} <Rm>, <Rs> // Outside IT block, and <Rd>, <Rm>, <type>, <Rs> can be represented in T1
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

```
RORS{<c>}{<q>} {<Rd>,} <Rm>, <Rs>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ROR <Rs>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rdm>`	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the first general-purpose source register, encoded in the "Rm" field.
`<Rs>`	Is the second general-purpose source register holding a rotate amount in its bottom 8 bits, encoded in the "Rs" field.

Operation for all encodings

The description of MOV, MOVS (register-shifted register) gives the operational pseudocode for this instruction.

C2.4.133 RRX

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

15 14 13 12 11 10 9	8 7 6 5	4	3 2 1 0	15	14 12	11 8	7 6	5 4	3 0
1 1 1 0 1 0 1	0 0 1 0	0	1 1 1 1	(0)	0 0 0	Rd	0 0	1 1	Rm
		S			imm3		imm2	type	

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

The description of MOV (register) gives the operational pseudocode for this instruction.

C2.4.134 RRXS

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	(0)	0	0	0	Rd	0	0	1	1	Rm
											S						imm3				imm2		type		

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

The description of MOV (register) gives the operational pseudocode for this instruction.

C2.4.135 RSB (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

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

T1 variant

```
RSB<c>{<q>} {<Rd>, }<Rn>, #0 // Inside IT block
RSBS{<q>} {<Rd>, }<Rn>, #0 // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = Zeros(32); // immediate = #0
```

T2

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	0	1 1 1 0	S	Rn	0	imm3	Rd	imm8

RSB variant

Applies when S == 0.

```
RSB<c>.W {<Rd>,} <Rn>, #0 // Inside IT block
RSB{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

RSBS variant

Applies when S == 1.

```
RSBS.W {<Rd>,} <Rn>, #0 // Outside IT block
RSBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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 *Modified immediate constants* on page C1-318 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;
```

C2.4.136 RSB (register)

Reverse Subtract (register) subtracts a register value from an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	1	1	1	0	S	Rn	(0)	imm3	Rd	imm2	type	Rm

RSB, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
RSB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

RSBS, rotate right with extend variant

Applies when `S == 1 && imm3 == 000 && imm2 == 00 && type == 11`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

RSBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && type == 11)`.

```
RSBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`

	`ROR` when `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

```
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;
```

C2.4.137 SADD16

Signed Add 16 performs two 16-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d] = sum2<15:0> : sum1<15:0>;
    APSR.GE<1:0> = if sum1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

C2.4.138 SADD8

Signed Add 8 performs four 8-bit signed integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d] = sum4<7:0> : sum3<7:0> : sum2<7:0> : sum1<7:0>;
    APSR.GE<0>  = if sum1 >= 0 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0 then '1' else '0';
```

C2.4.139 SASX

Signed Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer addition and one 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d] = sum<15:0> : diff<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0 then '11' else '00';
```

C2.4.140 SBC (immediate)

Subtract with Carry (immediate) subtracts an immediate value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14 … 12	11 … 8	7 … 0
1	1	1	1	0	i	0	1	0	1	1	S	Rn	0	imm3	Rd	imm8

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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

C2.4.141 SBC (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5 3	2 0
0	1	0	0	0	0	0	1	1	0	Rm	Rdn

T1 variant

```
SBC<c>{<q>} {<Rdn>,} <Rdn>, <Rm> // Inside IT block
SBCS{<q>} {<Rdn>,} <Rdn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rdn);  n = UInt(Rdn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	1	0	1	1	S	Rn	(0)	imm3	Rd	imm2	type	Rm

SBC, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
SBC{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBC, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
SBCS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SBCS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

`LSL`	when `type = 00`
`LSR`	when `type = 01`
`ASR`	when `type = 10`
`ROR`	when `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;
```

C2.4.142 SBFX

Signed Bit Field Extract extracts any number of adjacent bits at any position from one register, sign extends them to 32 bits, and writes the result to the destination register.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9 8	7 6	5	4	3 0	15	14 12	11 8	7 6	5	4 0
1 1 1 1 0	(0)	1 1	0 1	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

T1 variant

```
SBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if msbit > 31 then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit <= 31 then
        R[d] = SignExtend(R[n]<msbit:lsbit>, 32);
    else
        R[d] = bits(32) UNKNOWN;
```

C2.4.143 SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value and writes the result to the destination register. The condition code flags are not affected.

Note

If R[n] == `0x80000000` (-2^{31}) and R[m] == `0xFFFFFFFF` (-1), the result of the division is `0x80000000`.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 8	7	6	5	4	3 … 0
1	1	1	1	1	0	1	1	1	0	0	1	Rn	(1)	(1)	(1)	(1)	Rd	1	1	1	1	Rm

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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>;
```

C2.4.144 SEL

Select Bytes selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SEL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<7:0>   = if APSR.GE<0> == '1' then R[n]<7:0>   else R[m]<7:0>;
    result<15:8>  = if APSR.GE<1> == '1' then R[n]<15:8>  else R[m]<15:8>;
    result<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    result<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
    R[d] = result;
```

C2.4.145 SEV

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. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    SendEvent();
```

C2.4.146 SG

Secure Gateway marks a valid branch target for branches from Non-secure code that call Secure code.

This instruction sets the Security state to Secure if its address is in Secure memory. If the address of this instruction is in Non-secure memory, the instruction behaves as a NOP.

If the PE was previously in Non-secure state:

- This instruction sets bit[0] of LR to 0, to indicate that the return address will cause a transition from Secure to Non-secure state.
- If the Floating-point Extension is implemented, this instruction marks Secure floating-point state as inactive, by setting CONTROL_S.SFPA to 0. This indicates that the floating-point registers do not contain active state that belongs to the Secure state.

If the Security Extension is not implemented, this instruction behaves as a NOP.

Note

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

`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation

```
EncodingSpecificOperations();

if HaveSecurityExt() then
    sAttributes = SecurityCheck(ThisInstrAddr(), TRUE, IsSecure());
    if !sAttributes.ns then
        if !IsSecure() then
            LR<0> = '0';
            if HaveFPExt() then
                CONTROL_S.SFPA = '0';
        CurrentState = SecurityState_Secure;
        // IT/ICI bits cleared to prevent Non-secure code interfering with
        // Secure execution
        if HaveMainExt() then
            ITSTATE = Zeros(8);
```

C2.4.147 SHADD16

Signed Halving Add 16 performs two signed 16-bit integer additions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d] = sum2<16:1> : sum1<16:1>;
```

C2.4.148 SHADD8

Signed Halving Add 8 performs four signed 8-bit integer additions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d] = sum4<8:1> : sum3<8:1> : sum2<8:1> : sum1<8:1>;
```

C2.4.149 SHASX

Signed Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer addition and one signed 16-bit subtraction, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

15	14	13	12	11	8	7	6	5	4	3	0
1	1	1	1	Rd		0	0	1	0	Rm	

T1 variant

```
SHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d] = sum<16:1> : diff<16:1>;
```

C2.4.150 SHSAX

Signed Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one signed 16-bit integer subtraction and one signed 16-bit addition, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d] = diff<16:1> : sum<16:1>;
```

C2.4.151 SHSUB16

Signed Halving Subtract 16 performs two signed 16-bit integer subtractions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]  = diff2<16:1> : diff1<16:1>;
```

C2.4.152 SHSUB8

Signed Halving Subtract 8 performs four signed 8-bit integer subtractions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]  = diff4<8:1> : diff3<8:1> : diff2<8:1> : diff1<8:1>;
```

C2.4.153 SMLABB, SMLABT, SMLATB, SMLATT

Signed Multiply Accumulate (halfwords) performs a signed multiply accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the APSR. It is not possible for overflow to occur during the multiplication.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1	1	1	1	1	0	1	1	0	0	0	1	Rn	!=1111	Rd	0	0	N	M	Rm
													Ra						

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 all variants of this encoding

```
if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
n_high = (N == '1');  m_high = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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';
```

C2.4.154 SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	0	0	1	0	Rn	!=1111	Rd	0	0	0	M	Rm
													Ra						

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 all variants of this encoding

```
if Ra == '1111' then SEE SMUAD;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
m_swap = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    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[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

C2.4.155 SMLAL

Signed Multiply Accumulate Long multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	0	0	Rn	RdLo	RdHi	0	0	0	0	Rm

T1 variant

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

C2.4.156 SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	0	0	Rn	RdLo	RdHi	1	0	N	M	Rm

SMLALBB variant

Applies when `N == 0 && M == 0`.

```
SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALBT variant

Applies when `N == 0 && M == 1`.

```
SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTB variant

Applies when `N == 1 && M == 0`.

```
SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALTT variant

Applies when `N == 1 && M == 1`.

```
SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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>;
```

C2.4.157 SMLALD, SMLALDX

Signed Multiply Accumulate Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	0	0	Rn	RdLo	RdHi	1	1	0	M	Rm

SMLALD variant

Applies when `M == 0`.

```
SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

SMLALDX variant

Applies when `M == 1`.

```
SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    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>;
```

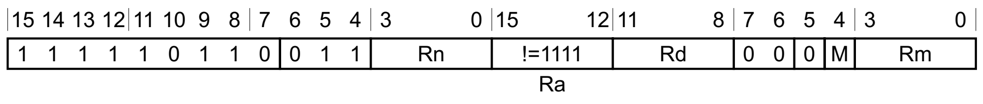
C2.4.158 SMLAWB, SMLAWT

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 all variants of this encoding

```
if Ra == '1111' then SEE "SMULWB, SMULWT";
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_high = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    result = SInt(R[n]) * SInt(operand2) + (SInt(R[a]) << 16);
    R[d] = result<47:16>;
    if (result >> 16) != SInt(R[d]) then  // Signed overflow
        APSR.Q = '1';
```

C2.4.159 SMLSD, SMLSDX

Signed Multiply Subtract Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7	6	5	4	3 … 0
1	1	1	1	1	0	1	1	0	1	0	0	Rn	!=1111	Rd	0	0	0	M	Rm
													Ra						

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 all variants of this encoding

```
if Ra == '1111' then SEE SMUSD;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  m_swap = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    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[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

C2.4.160 SMLSLD, SMLSLDX

Signed Multiply Subtract Long Dual performs two signed 16-bit by 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	0	1	Rn	RdLo	RdHi	1	1	0	M	Rm

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 all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    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>;
```

C2.4.161 SMMLA, SMMLAR

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

15 14 13 12 11 10 9 8 7	6 5 4	3 0	15 12	11 8	7 6	5	4	3 0
1 1 1 1 1 0 1 1 0	1 0 1	Rn	!=1111	Rd	0 0	0	R	Rm
			Ra					

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 all variants of this encoding

```
if Ra == '1111' then SEE SMMUL;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

C2.4.162 SMMLS, SMMLSR

Signed Most Significant Word Multiply Subtract multiplies two signed 32-bit values, subtracts the result from a 32-bit accumulate value that is shifted left by 32 bits, and extracts the most significant 32 bits of the result of that subtraction.

Optionally, the instruction can specify that the result of the instruction is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the result of the subtraction before the high word is extracted.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	0	1	1	0	Rn	Ra	Rd	0	0	0	R	Rm

SMMLS variant

Applies when `R == 0`.

```
SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

SMMLSR variant

Applies when `R == 1`.

```
SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);  round = (R == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

C2.4.163 SMMUL, SMMULR

Signed Most Significant Word Multiply multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and writes those bits to the destination register.

Optionally, the instruction can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

T1

Armv8-M DSP Extension only

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

SMMUL variant

Applies when `R == 0`.

```
SMMUL{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMMULR variant

Applies when `R == 1`.

```
SMMULR{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  round = (R == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

C2.4.164 SMUAD, SMUADX

Signed Dual Multiply Add performs two signed 16-bit by 16-bit multiplications. It adds the products together, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

T1

Armv8-M DSP Extension only

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

SMUAD variant

Applies when `M == 0`.

```
SMUAD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUADX variant

Applies when `M == 1`.

```
SMUADX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    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;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then  // Signed overflow
        APSR.Q = '1';
```

C2.4.165 SMULBB, SMULBT, SMULTB, SMULTT

Signed Multiply (halfwords) multiplies two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is written to the destination register. No overflow is possible during this instruction.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	0	0	0	1	Rn	1	1	1	1	Rd	0	0	N	M	Rm

SMULBB variant

Applies when `N == 0 && M == 0`.

```
SMULBB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULBT variant

Applies when `N == 0 && M == 1`.

```
SMULBT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTB variant

Applies when `N == 1 && M == 0`.

```
SMULTB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULTT variant

Applies when `N == 1 && M == 1`.

```
SMULTT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);
n_high = (N == '1');  m_high = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

C2.4.166 SMULL

Signed Multiply Long multiplies two 32-bit signed values to produce a 64-bit result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	0	0	0	Rn	RdLo	RdHi	0	0	0	0	Rm

T1 variant

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = SInt(R[n]) * SInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

C2.4.167 SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

T1

Armv8-M DSP Extension only

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

SMULWB variant

Applies when `M == 0`.

```
SMULWB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMULWT variant

Applies when `M == 1`.

```
SMULWT{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_high = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

C2.4.168 SMUSD, SMUSDX

Signed Dual Multiply Subtract performs two signed 16-bit by 16-bit multiplications. It subtracts one of the products from the other, and writes the result to the destination register.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic.

Overflow cannot occur.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 8	7	6	5	4	3 … 0
1	1	1	1	1	0	1	1	0	1	0	0	Rn	1	1	1	1	Rd	0	0	0	M	Rm

SMUSD variant

Applies when `M == 0`.

```
SMUSD{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

SMUSDX variant

Applies when `M == 1`.

```
SMUSDX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for all variants of this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  m_swap = (M == '1');
if d IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    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;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

C2.4.169 SSAT

Signed Saturate saturates an optionally-shifted signed value to a selectable signed range.

The APSR.Q flag is set to 1 if the operation saturates.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5	4 0
1	1	1	1	0	(0)	1	1	0	0	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm

Arithmetic shift right variant

Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of this encoding

```
if sh == '1' && (imm3:imm2) == '00000' then
    if HaveDSPExt() then
        SEE SSAT16;
    else
        UNDEFINED;
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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>. 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand = Shift(R[n], shift_t, shift_n, APSR.C);  // APSR.C ignored
    (result, sat) = SignedSatQ(SInt(operand), saturate_to);
```

```
R[d] = SignExtend(result, 32);
if sat then
    APSR.Q = '1';
```

C2.4.170 SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

The APSR.Q flag is set to 1 if the operation saturates.

T1

Armv8-M DSP Extension only

15 14 13 12 11	10	9 8	7 6	5	4	3 0	15	14 13 12	11 8	7 6	5	4	3 0
1 1 1 1 0	(0)	1 1	0 0	1	0	Rn	0	0 0 0	Rd	0 0	(0)	(0)	sat_imm

T1 variant

```
SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm)+1;
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    bits(32) result;
    result<15:0>  = SignExtend(result1, 16);
    result<31:16> = SignExtend(result2, 16);
    R[d] = result;
    if sat1 || sat2 then
        APSR.Q = '1';
```

C2.4.171 SSAX

Signed Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one 16-bit integer subtraction and one 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d] = diff<15:0> : sum<15:0>;
    APSR.GE<1:0> = if sum  >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

C2.4.172 SSUB16

Signed Subtract 16 performs two 16-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]  = diff2<15:0> : diff1<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

C2.4.173 SSUB8

Signed Subtract 8 performs four 8-bit signed integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
SSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]  = diff4<7:0> : diff3<7:0> : diff2<7:0> : diff1<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

C2.4.174 STC, STC2

Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	0	1	1	0	P	U	D	W	0	Rn	CRd	!=101x	imm8
														coproc	

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 all variants of this encoding

```
if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
if coproc IN '101x' then SEE "Floating-point";
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;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	1	1	1	0	P	U	D	W	0	Rn	CRd	!=101x	imm8
														coproc	

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 all variants of this encoding

```
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 coproc IN '101x' 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;
```

Notes for all encodings

See Floating-point: *Floating-point load/store* on page C2-365.

Assembler symbols

`L`	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`-`	when `U = 0`
`+`	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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
        repeat
            MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());
            address         = address + 4;
        until Coproc_DoneStoring(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;
```

C2.4.175 STL

Store Release Word stores a word from a register to memory. The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
STL{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 4] = R[t];
```

C2.4.176 STLB

Store Release Byte stores a byte from a register to memory. The instruction also has memory ordering semantics.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	0	0	0	(1)	(1)	(1)	(1)

T1 variant

```
STLB{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 1] = R[t]<7:0>;
```

C2.4.177 STLEX

Store Release Exclusive Word stores a word from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	1	1	0	Rd

T1 variant

```
STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`0`	If the operation updates memory.
`1`	If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
```

```
if ExclusiveMonitorsPass(address,4) then
    MemO[address, 4] = R[t];
    R[d] = ZeroExtend('0');
else
    R[d] = ZeroExtend('1');
```

C2.4.178 STLEXB

Store Release Exclusive Byte stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	1	0	0	Rd

T1 variant

```
STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`0`	If the operation updates memory.
`1`	If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
```

```
        if ExclusiveMonitorsPass(address,1) then
            MemO[address, 1] = R[t]<7:0>;
            R[d] = ZeroExtend('0');
        else
            R[d] = ZeroExtend('1');
```

C2.4.179 STLEXH

Store Release Exclusive Halfword stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed. The instruction also has memory ordering semantics.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	1	1	0	1	Rd

T1 variant

```
STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:
	`0` If the operation updates memory.
	`1` If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
```

```
if ExclusiveMonitorsPass(address,2) then
    MemO[address, 2] = R[t]<15:0>;
    R[d] = ZeroExtend('0');
else
    R[d] = ZeroExtend('1');
```

C2.4.180 STLH

Store Release Halfword stores a halfword from a register to memory. The instruction also has memory ordering semantics.

T1

Armv8-M

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

T1 variant

```
STLH{<c>}{<q>} <Rt>, [<Rn>]
```

Decode for this encoding

```
t = UInt(Rt); n = UInt(Rn);
if t IN {13,15} || n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```

C2.4.181 STM, STMIA, STMEA

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

15	14	13	12	11	10 8	7 0
1	1	0	0	0	Rn	register_list

T1 variant

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // Preferred syntax
STMEA{<c>}{<q>} <Rn>!, <registers> // Alternate syntax, Empty Ascending stack
```

Decode for this encoding

```
n = UInt(Rn);  registers = '00000000':register_list;  wback = TRUE;
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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12 0
1	1	1	0	1	0	0	0	1	0	W	0	Rn	(0)	M	(0)	register_list

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.

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.

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

`IA`	Is an optional suffix for the Increment After form.
`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
        doOperation         = (!applylimit || (UInt(endAddress) >= UInt(limit)));
    else
        doOperation         = TRUE;

    for i = 0 to 14
        // Memory operation only performed if limit not violated
        if registers<i> == '1' && doOperation 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;
```

C2.4.182 STMDB, STMFD

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). The alias is always the preferred disassembly.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12 0
1	1	1	0	1	0	0	1	0	0	W	0	Rn	(0)	M	(0)	register_list

T1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // Preferred syntax
STMFD{<c>}{<q>} <Rn>{!}, <registers> // Alternate syntax, Full Descending stack
```

Decode for this encoding

```
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.

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.

If `BitCount(registers) == 1`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction operates as an `STM` with the same addressing mode but targeting an unspecified set of registers. These registers might include R15.

T2

Armv8-M

15	14	13	12	11	10	9	8	7 … 0
1	0	1	1	0	1	0	M	register_list

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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. 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);
    if n == 13 && wback then
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    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 || (UInt(address) >= UInt(limit))) 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);
```

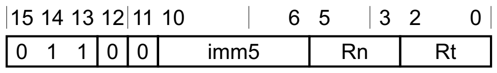
C2.4.183 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

This instruction is used by the alias PUSH (single register). See *Alias conditions* on page C2-669 for details of when each alias is preferred.

T1

Armv8-M



T1 variant

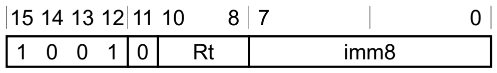
```
STR{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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

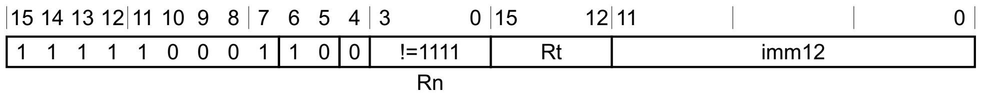
```
STR{<c>}{<q>} <Rt>, [SP{, #{+}<imm>}]
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7 … 0
1	1	1	1	1	0	0	0	0	1	0	0	!=1111	Rt	1	P	U	W	imm8
												Rn						

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 all variants of this encoding

```
if P == '1' && U == '1' && W == '0' then SEE STRT;
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 == 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	is preferred when
PUSH (single register)	`Rn == '1101' && U == '0' && imm8 == '00000100'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

-	when `U = 0`
+	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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.184 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M

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

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8 7 6	5 4	3 0
1 1 1 1 1 0 0 0 0	1 0	0	!=1111	Rt	0 0 0 0 0 0	imm2	Rm
			Rn				

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

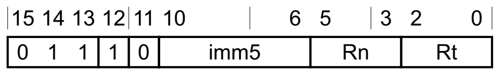
```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,4] = R[t];
```

C2.4.185 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M



T1 variant

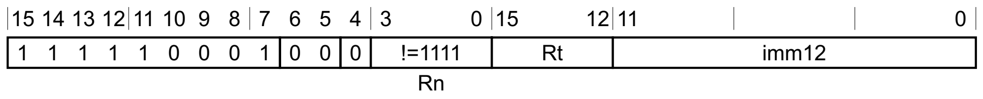
```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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

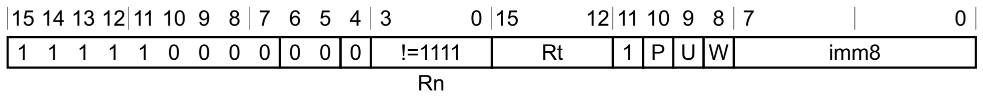
```
STRB{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // <Rt>, <Rn>, <imm> can be represented in T1
STRB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

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 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 all variants of 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`-`	when `U = 0`
`+`	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 an optional 5-bit unsigned immediate byte offset, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field.

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.

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
    (limit, applylimit) = LookUpSPLim(LookUpSP());
else
    applylimit = FALSE;
// Memory operation only performed if limit not violated
if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
    MemU[address,1] = R[t]<7:0>;

// If the stack pointer is being updated a fault will be raised if
// the limit is violated
if wback then RSPCheck[n] = offset_addr;
```

C2.4.186 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M

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

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5 4	3 0
1	1	1	1	1	0	0	0	0	0	0	0	!=1111	Rt	0	0	0	0	0	0	imm2	Rm
												Rn									

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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>;
```

C2.4.187 STRBT

Store Register Byte Unprivileged calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. When privileged software uses an `STRBT` instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	1	1	0	0	0	0	0	0	0	!=1111	Rt	1	1	1	0	imm8
												Rn						

T1 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,1] = R[t]<7:0>;
```

C2.4.188 STRD (immediate)

Store Register Dual (immediate) calculates an address from a base register value and an immediate offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7 0
1	1	1	0	1	0	0	P	U	1	W	0	!=1111	Rt	Rt2	imm8
												Rn			

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 all variants of this encoding

```
if P == '0' && W == '0' then SEE "Related encodings";
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  t2 = UInt(Rt2);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1');  add = (U == '1');  wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t IN {13,15} || t2 IN {13,15} then UNPREDICTABLE;
```

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.

Notes for all encodings

Related encodings: *Load/store (multiple, dual, exclusive, acquire-release), table branch* on page C2-336.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:
	`-` when `U = 0`
	`+` 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

```
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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;
    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(offset_addr) >= UInt(limit)) 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;
```

C2.4.189 STREX

Store Register Exclusive calculates an address from a base register value and an immediate offset, and stores a word from a register to memory if the executing PE has exclusive access to the memory addressed.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11 … 8	7 … 0
1	1	1	0	1	0	0	0	0	1	0	0	Rn	!=1111	Rd	imm8
													Rt		

T1 variant

```
STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 then SEE "TT";
if d IN {13,15} || t == 13 || n == 15 then UNPREDICTABLE;
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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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: `0` If the operation updates memory. `1` If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    if ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

C2.4.190 STREXB

Store Register Exclusive Byte derives an address from a base register value, and stores a byte from a register to memory if the executing PE has exclusive access to the memory addressed.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	0	0	0	1	1	0	0	Rn	Rt	(1)	(1)	(1)	(1)	0	1	0	0	Rd

T1 variant

```
STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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: `0` If the operation updates memory. `1` If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
```

```
if ExclusiveMonitorsPass(address,1) then
    MemA[address,1] = R[t]<7:0>;
    R[d] = ZeroExtend('0');
else
    R[d] = ZeroExtend('1');
```

C2.4.191 STREXH

Store Register Exclusive Halfword derives an address from a base register value, and stores a halfword from a register to memory if the executing PE has exclusive access to the memory addressed.

T1

Armv8-M

15 14 13 12 11 10 9 8 7 6 5	4	3 0	15 12	11 10 9 8	7 6	5 4	3 0
1 1 1 0 1 0 0 0 1 1 0	0	Rn	Rt	(1) (1) (1) (1)	0 1	0 1	Rd

T1 variant

```
STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]
```

Decode for this encoding

```
d = UInt(Rd);  t = UInt(Rt);  n = UInt(Rn);
if d IN {13,15} || t IN {13,15} || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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.

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:

`0`	If the operation updates memory.
`1`	If the operation fails to update 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
```

```
if ExclusiveMonitorsPass(address,2) then
    MemA[address,2] = R[t]<15:0>;
    R[d] = ZeroExtend('0');
else
    R[d] = ZeroExtend('1');
```

C2.4.192 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. It can use offset, post-indexed, or pre-indexed addressing.

T1

Armv8-M

15 14 13 12	11	10 6	5 3	2 0
1 0 0 0	0	imm5	Rn	Rt

T1 variant

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

T2

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 0
1 1 1 1 1 0 0 0 1	0 1	0	!=1111	Rt	imm12
			Rn		

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

```
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 IN {13,15} then UNPREDICTABLE;
```

T3

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11	10	9	8	7 0
1 1 1 1 1 0 0 0 0	0 1	0	!=1111	Rt	1	P	U	W	imm8
			Rn						

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 all variants of 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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:
	`-` when `U = 0`
	`+` 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 2 in the range 0 to 62, defaulting to 0 and encoded in the "imm5" field as <imm>/2.
	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.
	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];
```

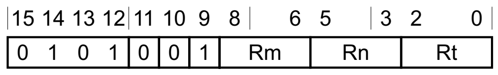
```
// Determine if the stack pointer limit should be checked
if n == 13 && wback then
    (limit, applylimit) = LookUpSPLim(LookUpSP());
else
    applylimit = FALSE;
// Memory operation only performed if limit not violated
if !applylimit || (UInt(offset_addr) >= UInt(limit)) then
    MemU[address,2] = R[t]<15:0>;

// If the stack pointer is being updated a fault will be raised if
// the limit is violated
if wback then RSPCheck[n] = offset_addr;
```

C2.4.193 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. The offset register value can be shifted left by 0, 1, 2, or 3 bits.

T1

Armv8-M



T1 variant

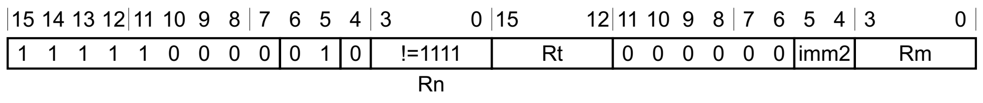
```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

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

```
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

```
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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,2] = R[t]<15:0>;
```

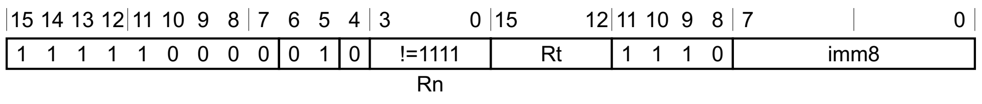
C2.4.194 STRHT

Store Register Halfword Unprivileged calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory.

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

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    MemU_unpriv[address,2] = R[t]<15:0>;
```

C2.4.195 STRT

Store Register Unprivileged calculates an address from a base register value and an immediate offset, and stores a word from a register to memory.

When privileged software uses an STRT instruction, the memory access is restricted as if the software was unprivileged.

T1

Armv8-M Main Extension only

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 12	11 10 9 8	7 0
1 1 1 1 1 0 0 0 0	1 0	0	!=1111	Rt	1 1 1 0	imm8
			Rn			

T1 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
if !HaveMainExt() then UNDEFINED;
t = UInt(Rt);  n = UInt(Rn);  postindex = FALSE;  add = TRUE;
register_form = FALSE;  imm32 = ZeroExtend(imm8, 32);
if t IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] + imm32;
    data = R[t];
    MemU_unpriv[address,4] = data;
```

C2.4.196 SUB (SP minus immediate)

Subtract (SP minus immediate) subtracts an immediate value from the SP value, and writes the result to the destination register.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6 0
1	0	1	1	0	0	0	0	1	imm7

T1 variant

```
SUB{<c>}{<q>} {SP,} SP, #<imm7>
```

Decode for this encoding

```
d = 13;  setflags = FALSE;  imm32 = ZeroExtend(imm7:'00', 32);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	1	0	1	S	1	1	0	1	0	imm3	Rd	imm8

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 all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = (S == '1');  imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && S == '0' then UNPREDICTABLE;
```

T3

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 0
1	1	1	1	0	i	1	0	1	0	1	0	1	1	0	1	0	imm3	Rd	imm8

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

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  setflags = FALSE;  imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    RSPCheck[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

C2.4.197 SUB (SP minus 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14 12	11 8	7 6	5 4	3 0
1	1	1	0	1	0	1	1	1	0	1	S	1	1	0	1	(0)	imm3	Rd	imm2	type	Rm

SUB, rotate right with extend variant

Applies when `S == 0 && imm3 == 000 && imm2 == 00 && type == 11`.

```
SUB{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm>, RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && type == 11) && Rd != 1111`.

```
SUBS{<c>}{<q>} {<Rd>,} SP, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if d == 13 && (shift_t != SRType_LSL || shift_n > 3) then UNPREDICTABLE;
if (d == 15 && S == '0') || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when `type = 00`
LSR	when `type = 01`

ASR	when type = 10
ROR	when 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

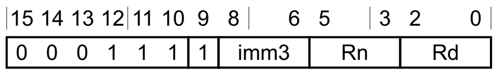
```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, NOT(shifted), '1');
    RSPCheck[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

C2.4.198 SUB (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

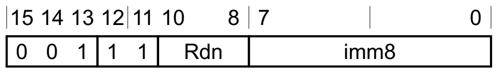
```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // Inside IT block
SUBS{<q>} <Rd>, <Rn>, #<imm3> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

T2

Armv8-M



T2 variant

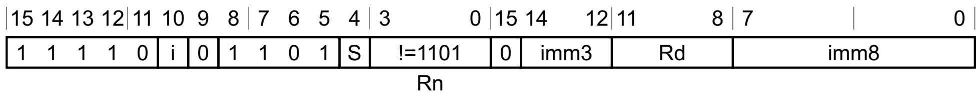
```
SUB<c>{<q>} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> can be represented in T1
SUB<c>{<q>} {<Rdn>,} <Rdn>, #<imm8> // Inside IT block, and <Rdn>, <imm8> cannot be represented in T1
SUBS{<q>} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> can be represented in T1
SUBS{<q>} {<Rdn>,} <Rdn>, #<imm8> // Outside IT block, and <Rdn>, <imm8> cannot be represented in T1
```

Decode for this encoding

```
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 all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus 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

15 14 13 12 11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1	0	1	0	1	0	!=11x1	0	imm3	Rd	imm8
								Rn				

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

```
if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE "SUB (SP minus 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rdn>`	Is the general-purpose source and destination register, encoded in the "Rdn" field.
`<imm8>`	Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.
`<Rn>`	For encoding T1: is the general-purpose source register, encoded in the "Rn" field. For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB (SP minus immediate). For encoding T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB (SP minus immediate). If the PC is used, see ADR.
`<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 *Modified immediate constants* on page C1-318 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;
```

C2.4.199 SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias.

This instruction is an alias of the ADR instruction. This means that:

- The encodings in this description are named to match the encodings of ADR.
- The description of ADR gives the operational pseudocode for this instruction.

T2

Armv8-M Main Extension only

15 14 13 12 11	10	9 8	7	6	5	4	3 2 1 0	15	14 12	11 8	7 0
1 1 1 1 0	i	1 0	1	0	1	0	1 1 1 1	0	imm3	Rd	imm8

T2 variant

```
SUB{<c>}{<q>} <Rd>, PC, #<imm12>
```

is equivalent to

```
ADR{<c>}{<q>} <Rd>, <label>
```

and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<label>`	For encoding T1: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. Permitted values of the size of the offset are multiples of 4 in the range 0 to 1020. For encoding T2 and T3: the label of an instruction or literal data item whose address is to be loaded into `<Rd>`. The assembler calculates the required value of the offset from the `Align(PC, 4)` value of the `ADR` instruction to this label. If the offset is zero or positive, encoding T3 is used, with `imm32` equal to the offset. If the offset is negative, encoding T2 is used, with `imm32` equal to the size of the offset. That is, the use of encoding T2 indicates that the required offset is minus the value of `imm32`. Permitted values of the size of the offset are 0-4095.
`<imm12>`	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

Operation

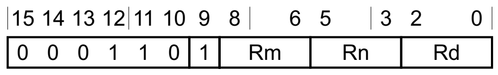
The description of ADR gives the operational pseudocode for this instruction.

C2.4.200 SUB (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

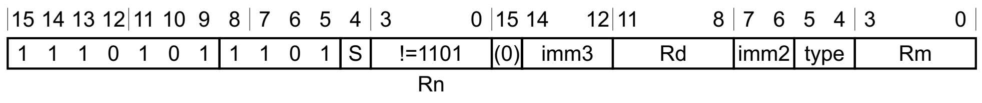
```
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // Inside IT block
SUBS{<q>} {<Rd>,} <Rn>, <Rm> // Outside IT block
```

Decode for this encoding

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(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 && type == 11`.

```
SUB{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value variant

Applies when `S == 0 && !(imm3 == 000 && imm2 == 00 && 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 && type == 11`.

```
SUBS{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value variant

Applies when `S == 1 && !(imm3 == 000 && imm2 == 00 && 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 all variants of 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(type, imm3:imm2);
if d == 13 || (d == 15 && S == '0') || n == 15 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as `<Rn>`.
`<Rn>`	For encoding T1: is the first general-purpose source register, encoded in the "Rn" field. For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see SUB (SP minus register).
`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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;
```

C2.4.201 SVC

The Supervisor Call instruction generates a call to a system supervisor.

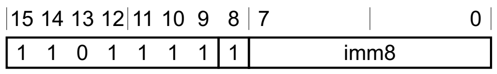
Use it as a call to an operating system to provide a service.

Note

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<imm>`	Is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

C2.4.202 SXTAB

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 14 13 12	11 8	7	6	5 4	3 0
1 1 1 1 1 0 1 0 0	1 0	0	!=1111	1 1 1 1	Rd	1	(0)	rotate	Rm
			Rn						

T1 variant

```
SXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE SXTB;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

C2.4.203 SXTAB16

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

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	1	1	0	1	0	0	0	1	0	!=1111
												Rn

15	14	13	12	11 … 8	7	6	5 4	3 … 0
1	1	1	1	Rd	1	(0)	rotate	Rm

T1 variant

```
SXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE SXTB16;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    bits(32) result;
    result<15:0>  = R[n]<15:0>  + SignExtend(rotated<7:0>,   16);
    result<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);
    R[d] = result;
```

C2.4.204 SXTAH

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

15	14	13	12	11	10	9	8	7	6	5	4	3 - 0	15	14	13	12	11 - 8	7	6	5 - 4	3 - 0
1	1	1	1	1	0	1	0	0	0	0	0	!=1111	1	1	1	1	Rd	1	(0)	rotate	Rm
												Rn									

T1 variant

```
SXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE SXTH;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

C2.4.205 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

T1

Armv8-M

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

T1 variant

```
SXTB{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

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

```
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`0`	when `rotate = 00`
`8`	when `rotate = 01`
`16`	when `rotate = 10`
`24`	when `rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

C2.4.206 SXTB16

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	0	1	0	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

T1 variant

```
SXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if !HaveDSPExt() 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`0`	`when rotate = 00`
`8`	`when rotate = 01`
`16`	`when rotate = 10`
`24`	`when rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    bits(32) result;
    result<15:0>  = SignExtend(rotated<7:0>,   16);
    result<31:16> = SignExtend(rotated<23:16>, 16);
    R[d] = result;
```

C2.4.207 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

T1

Armv8-M

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

T1 variant

```
SXTH{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	0	0	0	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

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

```
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`0`	`when rotate = 00`
`8`	`when rotate = 01`
`16`	`when rotate = 10`
`24`	`when rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

C2.4.208 TBB, TBH

Table Branch Byte causes a PC-relative forward branch using a table of single byte offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the byte returned from the table.

Table Branch Halfword causes a PC-relative forward branch using a table of single halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value of the halfword returned from the table.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	0	0	0	1	1	0	1	Rn	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm

Byte variant

Applies when `H == 0`.

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // Outside or last in IT block
```

Halfword variant

Applies when `H == 1`.

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // Outside or last in IT block
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);  is_tbh = (H == '1');
if n == 13 || m IN {13,15} then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC + 2*halfwords);
```

C2.4.209 TEQ (immediate)

Test Equivalence (immediate) performs an exclusive OR operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 10 9 8	7 0
1 1 1 1 0	i	0	0 1 0 0	1	Rn	0	imm3	1 1 1 1	imm8

T1 variant

```
TEQ{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] EOR imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

C2.4.210 TEQ (register)

Test Equivalence (register) performs an exclusive OR operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11	10	9	8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	1	0	0	1	Rn	(0)	imm3	1	1	1	1	imm2	type	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && type == 11`.

`TEQ{<c>}{<q>} <Rn>, <Rm>, RRX`

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.

`TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}`

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>` See *Standard assembler syntax fields* on page C1-310.

`<q>` See *Standard assembler syntax fields* on page C1-310.

`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    APSR.N = result<31>;
```

```
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            // APSR.V unchanged
```

C2.4.211 TST (immediate)

Test (immediate) performs a logical AND operation on a register value and an immediate value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M Main Extension only

15 14 13 12 11	10	9	8 7 6 5	4	3 0	15	14 12	11 10 9 8	7 0
1 1 1 1 0	i	0	0 0 0 0	1	Rn	0	imm3	1 1 1 1	imm8

T1 variant

```
TST{<c>}{<q>} <Rn>, #<const>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, APSR.C);
if n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 *Modified immediate constants* on page C1-318 for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = R[n] AND imm32;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

C2.4.212 TST (register)

Test (register) performs a logical AND operation on a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

T1

Armv8-M

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

T1 variant

```
TST{<c>}{<q>} <Rn>, <Rm>
```

Decode for this encoding

```
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = (SRType_LSL, 0);
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11	10	9	8	7 6	5 4	3 0
1	1	1	0	1	0	1	0	0	0	0	1	Rn	(0)	imm3	1	1	1	1	imm2	type	Rm

Rotate right with extend variant

Applies when `imm3 == 000 && imm2 == 00 && type == 11`.

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value variant

Applies when `!(imm3 == 000 && imm2 == 00 && type == 11)`.

```
TST{<c>}.W <Rn>, <Rm> // <Rn>, <Rm> can be represented in T1
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);  m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(type, imm3:imm2);
if n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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 "type" field. It can have the following values:

LSL	when type = 00
LSR	when type = 01
ASR	when type = 10
ROR	when 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
```

C2.4.213 TT, TTT, TTA, TTAT

Test Target (TT) queries the Security state and access permissions of a memory location.

Test Target Unprivileged (TTT) queries the Security state and access permissions of a memory location for an unprivileged access to that location.

Test Target Alternate Domain (TTA) and Test Target Alternate Domain Unprivileged (TTAT) query the Security state and access permissions of a memory location for a Non-secure access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

These instructions return the Security state and access permissions in the destination register. See TT_RESP for the format of the destination register.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	0	0	1	0	0	Rn	1	1	1	1	Rd	A	T	(0)	(0)	(0)	(0)	(0)	(0)

TT variant

Applies when `A == 0 && T == 0`.

```
TT{<c>}{<q>} <Rd>, <Rn>
```

TTA variant

Applies when `A == 1 && T == 0`.

```
TTA{<c>}{<q>} <Rd>, <Rn>
```

TTAT variant

Applies when `A == 1 && T == 1`.

```
TTAT{<c>}{<q>} <Rd>, <Rn>
```

TTT variant

Applies when `A == 0 && T == 1`.

```
TTT{<c>}{<q>} <Rd>, <Rn>
```

Decode for all variants of this encoding

```
d = UInt(Rd); n = UInt(Rn); alt = (A == '1'); forceunpriv = (T == '1');
if d IN {13,15} || n == 15 then UNPREDICTABLE;
if alt && !IsSecure() then UNDEFINED;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    addr = R[n];
    R[d] = TTResp(addr, alt, forceunpriv);
```

C2.4.214 UADD16

Unsigned Add 16 performs two 16-bit unsigned integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d] = sum2<15:0> : sum1<15:0>;
    APSR.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

C2.4.215 UADD8

Unsigned Add 8 performs four unsigned 8-bit integer additions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the additions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d] = sum4<7:0> : sum3<7:0> : sum2<7:0> : sum1<7:0>;
    APSR.GE<0>  = if sum1 >= 0x100 then '1' else '0';
    APSR.GE<1>  = if sum2 >= 0x100 then '1' else '0';
    APSR.GE<2>  = if sum3 >= 0x100 then '1' else '0';
    APSR.GE<3>  = if sum4 >= 0x100 then '1' else '0';
```

C2.4.216 UASX

Unsigned Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d] = sum<15:0> : diff<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
    APSR.GE<3:2> = if sum  >= 0x10000 then '11' else '00';
```

C2.4.217 UBFX

Unsigned Bit Field Extract extracts any number of adjacent bits at any position from one register, zero extends them to 32 bits, and writes the result to the destination register.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5	4 0
1	1	1	1	0	(0)	1	1	1	1	0	0	Rn	0	imm3	Rd	imm2	(0)	widthm1

T1 variant

```
UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);
lsbit = UInt(imm3:imm2);  widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if msbit > 31 then UNPREDICTABLE;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if msbit <= 31 then
        R[d] = ZeroExtend(R[n]<msbit:lsbit>, 32);
    else
        R[d] = bits(32) UNKNOWN;
```

C2.4.218 UDF

Permanently Undefined generates an Undefined Instruction exception.

T1

Armv8-M

15	14	13	12	11	10	9	8	7 … 0
1	1	0	1	1	1	1	0	imm8

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

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15	14	13	12	11 … 0
1	1	1	1	0	1	1	1	1	1	1	1	imm4	1	0	1	0	imm12

T2 variant

```
UDF{<c>}.W {#}<imm> // <imm> can be represented in T1
UDF{<c>}{<q>} {#}<imm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310. Arm deprecates using any <c> value other than `AL`.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<imm>`	For encoding T1: is an 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field. The PE ignores the value of this constant. For encoding T2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field. The PE ignores the value of this constant.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

C2.4.219 UDIV

Unsigned Divide divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. The condition flags are not affected.

T1

Armv8-M

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	0	1	1	Rn	(1)	(1)	(1)	(1)	Rd	1	1	1	1	Rm

T1 variant

```
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if UInt(R[m]) == 0 then
        if IntegerZeroDivideTrappingEnabled() then
            GenerateIntegerZeroDivide();
        else
            result = 0;
    else
        result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
    R[d] = result<31:0>;
```

C2.4.220 UHADD16

Unsigned Halving Add 16 performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d] = sum2<16:1> : sum1<16:1>;
```

C2.4.221 UHADD8

Unsigned Halving Add 8 performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d] = sum4<8:1> : sum3<8:1> : sum2<8:1> : sum1<8:1>;
```

C2.4.222 UHASX

Unsigned Halving Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d] = sum<16:1> : diff<16:1>;
```

C2.4.223 UHSAX

Unsigned Halving Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d] = diff<16:1> : sum<16:1>;
```

C2.4.224 UHSUB16

Unsigned Halving Subtract 16 performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]  = diff2<16:1> : diff1<16:1>;
```

C2.4.225 UHSUB8

Unsigned Halving Subtract 8 performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UHSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]  = diff4<8:1> : diff3<8:1> : diff2<8:1> : diff1<8:1>;
```

C2.4.226 UMAAL

Unsigned Multiply Accumulate Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, adds two unsigned 32-bit values, and writes the 64-bit result to two registers.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	1	0	Rn	RdLo	RdHi	0	1	1	0	Rm

T1 variant

```
UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

C2.4.227 UMLAL

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	1	1	0	Rn	RdLo	RdHi	0	0	0	0	Rm

T1 variant

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

C2.4.228 UMULL

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11 8	7	6	5	4	3 0
1	1	1	1	1	0	1	1	1	0	1	0	Rn	RdLo	RdHi	0	0	0	0	Rm

T1 variant

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
dLo = UInt(RdLo);  dHi = UInt(RdHi);  n = UInt(Rn);  m = UInt(Rm);  setflags = FALSE;
if dLo IN {13,15} || dHi IN {13,15} || n IN {13,15} || m IN {13,15} then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = UInt(R[n]) * UInt(R[m]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

C2.4.229 UQADD16

Unsigned Saturating Add 16 performs two unsigned 16-bit integer additions, saturates the results to the 16-bit unsigned integer range 0 to 2^{16}-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQADD16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    bits(32) result;
    result<15:0>  = UnsignedSat(sum1, 16);
    result<31:16> = UnsignedSat(sum2, 16);
    R[d] = result;
```

C2.4.230 UQADD8

Unsigned Saturating Add 8 performs four unsigned 8-bit integer additions, saturates the results to the 8-bit unsigned integer range 0 to 2^8-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQADD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    bits(32) result;
    result<7:0>   = UnsignedSat(sum1, 8);
    result<15:8>  = UnsignedSat(sum2, 8);
    result<23:16> = UnsignedSat(sum3, 8);
    result<31:24> = UnsignedSat(sum4, 8);
    R[d] = result;
```

C2.4.231 UQASX

Unsigned Saturating Add and Subtract with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer addition and one unsigned 16-bit subtraction, saturates the results to the 16-bit unsigned integer range 0 to 2^{16}-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    bits(32) result;
    result<15:0>  = UnsignedSat(diff, 16);
    result<31:16> = UnsignedSat(sum,  16);
    R[d] = result;
```

C2.4.232 UQSAX

Unsigned Saturating Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, saturates the results to the 16-bit unsigned integer range 0 to 2^{16}-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQSAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    bits(32) result;
    result<15:0>  = UnsignedSat(sum,  16);
    result<31:16> = UnsignedSat(diff, 16);
    R[d] = result;
```

C2.4.233 UQSUB16

Unsigned Saturating Subtract 16 performs two unsigned 16-bit integer subtractions, saturates the results to the 16-bit unsigned integer range 0 to 2^{16}-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    bits(32) result;
    result<15:0>  = UnsignedSat(diff1, 16);
    result<31:16> = UnsignedSat(diff2, 16);
    R[d] = result;
```

C2.4.234 UQSUB8

Unsigned Saturating Subtract 8 performs four unsigned 8-bit integer subtractions, saturates the results to the 8-bit unsigned integer range 0 to 2^8-1, and writes the results to the destination register.

T1

Armv8-M DSP Extension only

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

T1 variant

```
UQSUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    bits(32) result;
    result<7:0>   = UnsignedSat(diff1, 8);
    result<15:8>  = UnsignedSat(diff2, 8);
    result<23:16> = UnsignedSat(diff3, 8);
    result<31:24> = UnsignedSat(diff4, 8);
    R[d] = result;
```

C2.4.235 USAD8

Unsigned Sum of Absolute Differences performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

T1

Armv8-M DSP Extension only

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

T1 variant

```
USAD8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

C2.4.236 USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

T1

Armv8-M DSP Extension only

15 14 13 12	11 10 9 8	7	6 5 4	3 0	15 12	11 8	7 6	5 4	3 0
1 1 1 1	1 0 1 1	0	1 1 1	Rn	!=1111	Rd	0 0	0 0	Rm
					Ra				

T1 variant

```
USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>
```

Decode for this encoding

```
if Ra == '1111' then SEE USAD8;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  a = UInt(Ra);
if d IN {13,15} || n IN {13,15} || m IN {13,15} || a == 13 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>)   - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>)  - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

C2.4.237 USAT

Unsigned Saturate saturates an optionally-shifted signed value to a selected unsigned range.

The Q flag is set to 1 if the operation saturates.

T1

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 6	5	4 0
1	1	1	1	0	(0)	1	1	1	0	sh	0	Rn	0	imm3	Rd	imm2	(0)	sat_imm

Arithmetic shift right variant

Applies when `sh == 1 && !(imm3 == 000 && imm2 == 00)`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left variant

Applies when `sh == 0`.

```
USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

Decode for all variants of 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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>. 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

```
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';
```

C2.4.238 USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

The Q flag is set to 1 if the operation saturates.

T1

Armv8-M DSP Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5	4	3 0
1	1	1	1	0	(0)	1	1	1	0	1	0	Rn	0	0	0	0	Rd	0	0	(0)	(0)	sat_imm

T1 variant

```
USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  saturate_to = UInt(sat_imm);
if d IN {13,15} || n IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    bits(32) result;
    result<15:0>  = ZeroExtend(result1, 16);
    result<31:16> = ZeroExtend(result2, 16);
    R[d] = result;
    if sat1 || sat2 then
        APSR.Q = '1';
```

C2.4.239 USAX

Unsigned Subtract and Add with Exchange exchanges the two halfwords of the second operand, performs one unsigned 16-bit integer subtraction and one unsigned 16-bit addition, and writes the results to the destination register. It sets the APSR.GE bits according to the results.

T1

Armv8-M DSP Extension only

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

T1 variant

```
USAX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d] = diff<15:0> : sum<15:0>;
    APSR.GE<1:0> = if sum  >= 0x10000 then '11' else '00';
    APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

C2.4.240 USUB16

Unsigned Subtract 16 performs two 16-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
USUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]  = diff2<15:0> : diff1<15:0>;
    APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
    APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

C2.4.241 USUB8

Unsigned Subtract 8 performs four 8-bit unsigned integer subtractions, and writes the results to the destination register. It sets the APSR.GE bits according to the results of the subtractions.

T1

Armv8-M DSP Extension only

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

T1 variant

```
USUB8{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Decode for this encoding

```
if !HaveDSPExt() then UNDEFINED;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]  = diff4<7:0> : diff3<7:0> : diff2<7:0> : diff1<7:0>;
    APSR.GE<0>  = if diff1 >= 0 then '1' else '0';
    APSR.GE<1>  = if diff2 >= 0 then '1' else '0';
    APSR.GE<2>  = if diff3 >= 0 then '1' else '0';
    APSR.GE<3>  = if diff4 >= 0 then '1' else '0';
```

C2.4.242 UXTAB

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

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	1	0	1	!=1111	1	1	1	1	Rd	1	(0)	rotate	Rm
												Rn									

T1 variant

```
UXTAB{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE UXTB;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

C2.4.243 UXTAB16

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 14 13 12	11 8	7	6	5 4	3 0
1 1 1 1 1 0 1 0 0	0 1	1	!=1111	1 1 1 1	Rd	1	(0)	rotate	Rm
			Rn						

T1 variant

```
UXTAB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE UXTB16;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    bits(32) result;
    result<15:0>  = R[n]<15:0>  + ZeroExtend(rotated<7:0>,   16);
    result<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);
    R[d] = result;
```

C2.4.244 UXTAH

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

15 14 13 12 11 10 9 8 7	6 5	4	3 0	15 14 13 12	11 8	7	6	5 4	3 0
1 1 1 1 1 0 1 0 0	0 0	1	!=1111	1 1 1 1	Rd	1	(0)	rotate	Rm
			Rn						

T1 variant

```
UXTAH{<c>}{<q>} {<Rd>,} <Rn>, <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if Rn == '1111' then SEE UXTH;
if !HaveDSPExt() then UNDEFINED;
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  rotation = UInt(rotate:'000');
if d IN {13,15} || n == 13 || m IN {13,15} then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

C2.4.245 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

T1

Armv8-M

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

T1 variant

```
UXTB{<c>}{<q>} {<Rd>,} <Rm>
```

Decode for this encoding

```
d = UInt(Rd);  m = UInt(Rm);  rotation = 0;
```

T2

Armv8-M Main Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

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

```
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

`0`	`when rotate = 00`
`8`	`when rotate = 01`
`16`	`when rotate = 10`
`24`	`when rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

C2.4.246 UXTB16

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

15 14 13 12 11 10 9 8 7	6 5	4	3 2 1 0	15 14 13 12	11 8	7	6	5 4	3 0
1 1 1 1 1 0 1 0 0	0 1	1	1 1 1 1	1 1 1 1	Rd	1	(0)	rotate	Rm

T1 variant

```
UXTB16{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}
```

Decode for this encoding

```
if !HaveDSPExt() 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the second general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

0	when rotate = 00
8	when rotate = 01
16	when rotate = 10
24	when rotate = 11

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    bits(32) result;
    result<15:0>  = ZeroExtend(rotated<7:0>,   16);
    result<31:16> = ZeroExtend(rotated<23:16>, 16);
    R[d] = result;
```

C2.4.247 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

T1

Armv8-M

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

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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11 8	7	6	5 4	3 0
1	1	1	1	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	Rd	1	(0)	rotate	Rm

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rd>`	Is the general-purpose destination register, encoded in the "Rd" field.
`<Rm>`	Is the general-purpose source register, encoded in the "Rm" field.
`<amount>`	Is the rotate amount, encoded in the "rotate" field. It can have the following values:

0	when `rotate = 00`
8	when `rotate = 01`
16	when `rotate = 10`
24	when `rotate = 11`

ROR #<amount> can be omitted, meaning a rotate amount of 0. This is the preferred disassembly for rotate == `0b00`.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

C2.4.248 VABS

Floating-point Absolute takes the absolute value of a single-precision or double-precision register, and places the result in the destination register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9 8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	size	1	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VABS{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VABS{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPAbs(D[m]);
    else
        S[d] = FPAbs(S[m]);
```

C2.4.249 VADD

Floating-point Add adds two single-precision or double-precision registers, and places the result in the destination register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	1	1	Vn	Vd	1	0	1	sz	N	0	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VADD{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VADD{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPAdd(D[n], D[m], TRUE);
    else
        S[d] = FPAdd(S[n], S[m], TRUE);
```

C2.4.250 VCMP

Floating-point Compare compares two registers, or one register and zero. It writes the result to the FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent `VMRS` instruction.

It raises an Invalid Operation exception only if either operand is a signaling NaN.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd	1	0	1	sz	0	1	M	0	Vm
																					E				

Single-precision scalar variant

Applies when `sz == 0`.

```
VCMP{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCMP{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
quiet_nan_exc = (E == '1');  with_zero = 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);
```

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd	1	0	1	sz	0	1	(0)	0	(0)	(0)	(0)	(0)
																					E							

Single-precision scalar variant

Applies when `sz == 0`.

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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();
    if dp_operation then
        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);
    else
        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);
```

C2.4.251 VCMPE

Floating-point Compare, raising Invalid Operation on NaN compares two registers, or one register and zero. It writes the result to the FPSCR condition flags. These are normally transferred to the APSR condition flags by a subsequent VMRS instruction.

It raises an Invalid Operation exception if either operand is any type of NaN.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd	1	0	1	sz	1	1	M	0	Vm
																					E				

Single-precision scalar variant

Applies when sz == 0.

```
VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when sz == 1.

```
VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
quiet_nan_exc = (E == '1');  with_zero = 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);
```

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd	1	0	1	sz	1	1	(0)	0	(0)	(0)	(0)	(0)
																					E							

Single-precision scalar variant

Applies when sz == 0.

```
VCMPE{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar variant

Applies when sz == 1.

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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();
    if dp_operation then
        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);
    else
        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);
```

C2.4.252 VCVT (between double-precision and single-precision)

This instruction does one of the following:

- Converts the value in a double-precision register to single-precision and writes the result to a single-precision register.
- Converts the value in a single-precision register to double-precision and writes the result to a double-precision register.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	1	1	M	0	Vm			

Encoding

Applies when `sz == 0`.

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Encoding

Applies when `sz == 1`.

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
CheckDecodeFaults(TRUE);
double_to_single = (sz == '1');
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();  ExecuteFPCheck();
    if double_to_single then
        S[d] = FPDoubleToSingle(D[m], TRUE);
    else
        D[d] = FPSingleToDouble(S[m], TRUE);
```

C2.4.253 VCVT (between floating-point and fixed-point)

Floating-point Convert (between floating-point and fixed-point) converts a value in a register from floating-point to fixed-point, or from fixed-point to floating-point, and places the result in the destination register. Software can specify the fixed-point value as either signed or unsigned.

The fixed-point value can be 16-bit or 32-bit. Conversions from fixed-point values take their operand from the low-order bits of the source register and ignore any remaining bits. Signed conversions to fixed-point values sign-extend the result value to the destination register width. Unsigned conversions to fixed-point values zero-extend the result value to the destination register width.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

T1

Armv8-M Floating-point Extension only, sf == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	1	op	1	U	Vd	1	0	1	sf	sx	1	i	0	imm4

Single-precision scalar variant

Applies when `op == 0 && sf == 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar variant

Applies when `op == 1 && sf == 0`.

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 0 && sf == 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar variant

Applies when `op == 1 && sf == 1`.

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

Decode for all variants of this encoding

```
dp_operation = (sf == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the fixed-point number, encoded in the "U:sx" field. It can have the following values:

`S16`	when `U = 0`, `sx = 0`
`S32`	when `U = 0`, `sx = 1`
`U16`	when `U = 1`, `sx = 0`
`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:

- If `<dt>` is `S16` or `U16`, `<fbits>` must be in the range 0-16. (16 - `<fbits>`) is encoded in `[imm4, i]`
- If `<dt>` is `S32` or `U32`, `<fbits>` must be in the range 1-32. (32 - `<fbits>`) is encoded in `[imm4, i]`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_fixed then
        if dp_operation then
            result = FPToFixed(D[d], size, frac_bits, unsigned, round_zero, TRUE);
            D[d] = if unsigned then ZeroExtend(result, 64) else SignExtend(result, 64);
        else
            result = FPToFixed(S[d], size, frac_bits, unsigned, round_zero, TRUE);
            S[d] = if unsigned then ZeroExtend(result, 32) else SignExtend(result, 32);
    else
        if dp_operation then
            D[d] = FixedToFP(D[d]<size-1:0>, 64, frac_bits, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[d]<size-1:0>, 32, frac_bits, unsigned, round_nearest, TRUE);
```

C2.4.254 VCVT (floating-point to integer)

Convert floating-point to integer with Round towards Zero converts a value in a register from floating-point to a 32-bit integer, using the Round towards Zero rounding mode, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2 1 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	1	1 0 x	Vd	1	0	1	sz	1	1	M	0	Vm
													opc2						op				

Single-precision scalar variant

Applies when `opc2 == 100 && sz == 0`.

```
VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && sz == 0`.

```
VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && sz == 1`.

```
VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && sz == 1`.

```
VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
to_integer = (opc2<2> == '1');
if to_integer then
    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
    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);
```

Notes for all encodings

Related encodings: VCVT (between floating-point and fixed-point).

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
        else
            S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
```

C2.4.255 VCVT (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 the FPSCR, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15–12	11	10	9	8	7	6	5	4	3–0
1	1	1	0	1	1	1	0	1	D	1	1	1	0	0	0	Vd	1	0	1	sz	op	1	M	0	Vm
													opc2												

Single-precision scalar variant

Applies when `sz == 0`.

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
to_integer = (opc2<2> == '1');
if to_integer then
    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
    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);
```

Notes for all encodings

Related encodings: VCVT (between floating-point and fixed-point).

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the operand, encoded in the "op" field. It can have the following values:

`U32`	when `op = 0`
`S32`	when `op = 1`

`<Sd>`	Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
`<Dd>`	Is the 64-bit name of the floating-point destination register, encoded in the "D:Vd" field.
`<Sm>`	Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
        else
            S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
```

C2.4.256 VCVTA

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0 0	Vd	1	0	1	sz	op	1	M	0	Vm
														RM										

Single-precision scalar variant

Applies when `sz == 0`.

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
unsigned = (op == '0');
round_mode = RM;
d = UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
	`U32` when `op = 0`
	`S32` when `op = 1`
`<Sd>`	Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
    S[d] = FPToFixedDirected(D[m],0,unsigned,round_mode,TRUE);
else
    S[d] = FPToFixedDirected(S[m],0,unsigned,round_mode,TRUE);
```

C2.4.257 VCVTB

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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 - 12	11	10	9	8	7	6	5	4	3 - 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	0	1	M	0	Vm
																					T				

Single-precision scalar variant

Applies when `op == 0 && sz == 0`.

```
VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `op == 1 && sz == 0`.

```
VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `op == 0 && sz == 1`.

```
VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Double-precision scalar variant

Applies when `op == 1 && sz == 1`.

```
VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
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);
```

C2.4.258 VCVTM

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd	1	0	1	sz	op	1	M	0	Vm
														RM											

Single-precision scalar variant

Applies when `sz == 0`.

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
unsigned = (op == '0');
round_mode = RM;
d = UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
	`U32` when `op = 0`
	`S32` when `op = 1`
`<Sd>`	Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

Operation

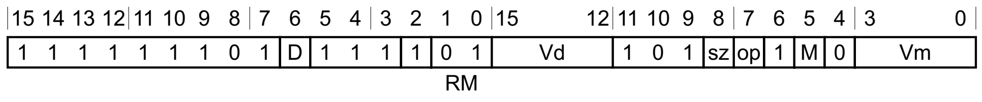
```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
    S[d] = FPToFixedDirected(D[m],0,unsigned,round_mode,TRUE);
else
    S[d] = FPToFixedDirected(S[m],0,unsigned,round_mode,TRUE);
```

C2.4.259 VCVTN

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.



Single-precision scalar variant

Applies when `sz == 0`.

```
VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
unsigned = (op == '0');
round_mode = RM;
d = UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
	`U32` when `op = 0`
	`S32` when `op = 1`
`<Sd>`	Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
    S[d] = FPToFixedDirected(D[m],0,unsigned,round_mode,TRUE);
else
    S[d] = FPToFixedDirected(S[m],0,unsigned,round_mode,TRUE);
```

C2.4.260 VCVTP

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0	Vd				1	0	1	sz	op	1	M	0	Vm			
														RM																	

Single-precision scalar variant

Applies when `sz == 0`.

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
unsigned = (op == '0');
round_mode = RM;
d = UInt(Vd:D);
m = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
```

Assembler symbols

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	Is the data type for the elements of the destination, encoded in the "op" field. It can have the following values:
	`U32` when `op = 0`
	`S32` when `op = 1`
`<Sd>`	Is the 32-bit name of the floating-point destination register, encoded in the "Vd:D" field.
`<Sm>`	Is the 32-bit name of the floating-point source register, encoded in the "Vm:M" field.
`<Dm>`	Is the 64-bit name of the floating-point source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
    S[d] = FPToFixedDirected(D[m],0,unsigned,round_mode,TRUE);
else
    S[d] = FPToFixedDirected(S[m],0,unsigned,round_mode,TRUE);
```

C2.4.261 VCVTR

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 the FPSCR, and places the result in the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15 14 13 12 11 10 9 8 7	6	5 4	3	2 0	15 12	11 10 9	8	7	6	5	4	3 0
1 1 1 0 1 1 1 0 1	D	1 1	1	1 0 x	Vd	1 0 1	sz	0	1	M	0	Vm
				opc2				op				

Single-precision scalar variant

Applies when `opc2 == 100 && sz == 0`.

```
VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `opc2 == 101 && sz == 0`.

```
VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `opc2 == 100 && sz == 1`.

```
VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>
```

Double-precision scalar variant

Applies when `opc2 == 101 && sz == 1`.

```
VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
if opc2 != '000' && !(opc2 IN '10x') then SEE "Related encodings";
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
to_integer = (opc2<2> == '1');
if to_integer then
    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
    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);
```

Notes for all encodings

Related encodings: VCVT (between floating-point and fixed-point).

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_integer then
        if dp_operation then
            S[d] = FPToFixed(D[m], 32, 0, unsigned, round_zero, TRUE);
        else
            S[d] = FPToFixed(S[m], 32, 0, unsigned, round_zero, TRUE);
    else
        if dp_operation then
            D[d] = FixedToFP(S[m], 64, 0, unsigned, round_nearest, TRUE);
        else
            S[d] = FixedToFP(S[m], 32, 0, unsigned, round_nearest, TRUE);
```

C2.4.262 VCVTT

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.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	1	op	Vd	1	0	1	sz	1	1	M	0	Vm
																					T				

Single-precision scalar variant

Applies when `op == 0 && sz == 0`.

```
VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>
```

Single-precision scalar variant

Applies when `op == 1 && sz == 0`.

```
VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `op == 0 && sz == 1`.

```
VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>
```

Double-precision scalar variant

Applies when `op == 1 && sz == 1`.

```
VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
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);
```

C2.4.263 VDIV

Floating-point Divide divides one floating-point value by another floating-point value and writes the result to a third floating-point register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	0	0	Vn	Vd	1	0	1	sz	N	0	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VDIV{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VDIV{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPDiv(D[n], D[m], TRUE);
    else
        S[d] = FPDiv(S[n], S[m], TRUE);
```

C2.4.264 VFMA

Floating-point Fused Multiply Accumulate multiplies two registers, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	1	D	1	0	Vn	Vd	1	0	1	sz	N	0	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
    else
        op32 = if op1_neg then FPNeg(S[n]) else S[n];
        S[d] = FPMulAdd(S[d], op32, S[m], TRUE);
```

C2.4.265 VFMS

Floating-point Fused Multiply Subtract negates one register and multiplies it with another register, adds the product to the destination register, and places the result in the destination register. The result of the multiply is not rounded before the addition.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	0	Vn	Vd	1	0	1	sz	N	1	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(D[d], op64, D[m], TRUE);
```

```
else
    op32 = if op1_neg then FPNeg(S[n]) else S[n];
    S[d] = FPMulAdd(S[d], op32, S[m], TRUE);
```

C2.4.266 VFNMA

Floating-point Fused Negate Multiply Accumulate negates one floating-point register value and multiplies it by another floating-point register value, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	0	1	Vn	Vd	1	0	1	sz	N	1	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
```

```
else
    op32 = if op1_neg then FPNeg(S[n]) else S[n];
    S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
```

C2.4.267 VFNMS

Floating-point Fused Negate Multiply Subtract multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register. The result of the multiply is not rounded before the addition.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	0	1	Vn	Vd	1	0	1	sz	N	0	M	0	Vm
																			op			

Single-precision scalar variant

Applies when sz == 0.

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when sz == 1.

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        op64 = if op1_neg then FPNeg(D[n]) else D[n];
        D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], TRUE);
```

```
else
    op32 = if op1_neg then FPNeg(S[n]) else S[n];
    S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], TRUE);
```

C2.4.268 VLDM

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. See *Alias conditions* on page C2-792 for details of when each alias is preferred.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 1	0
1	1	1	0	1	1	0	P	U	D	W	1	Rn	Vd	1	0	1	1	imm8<7:1>	0

imm8<0

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
CheckDecodeFaults();
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;
if n == 15 then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VLDM` with the same addressing mode but loads no registers.

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.

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.

T2

Armv8-M Floating-point Extension only

15 14 13 12 11 10 9	8	7	6	5	4	3 0	15 12	11 10 9	8	7 0
1 1 1 0 1 1 0	P	U	D	W	1	Rn	Vd	1 0 1	0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VLDR;
CheckDecodeFaults();
if P == '1' && U == '1' && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');
d = UInt(Vd:D);  n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8);
if n == 15 then UNPREDICTABLE;
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.

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.

Notes for all encodings

Related encodings: *Floating-point load/store and 64-bit register moves* on page C2-364.

Alias conditions

Alias	is preferred when
VPOP	`P == '0' && U == '1' && W == '1' && Rn == '1101'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
        // 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 applylimit && (UInt(address) < UInt(limit)) then
                if HaveMainExt() then
                    UFSR.STKOF = '1';
                // If Main Extension is not implemented the fault always escalates to HardFault
                excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
                HandleException(excInfo);

    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(regval) >= UInt(limit)) then
        for r = 0 to regs-1
            if single_regs then
                S[d+r]  = MemA[address,4];
                address = address+4;
            else
                word1   = MemA[address,4];  word2 = MemA[address+4,4];
                address = address+8;
                // Combine the word-aligned words in the correct order for
                // current endianness.
                D[d+r] = if BigEndian() then word1:word2 else word2:word1;

    // If the stack pointer is being updated a fault will be raised if
    // the limit is violated
    if wback then RSPCheck[n] = regval;
```

C2.4.269 VLDR

Floating-point Load Register loads a Floating-point Extension register from memory, using an address from a general-purpose register, with an optional offset.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	1	0	1	U	D	0	1	Rn	Vd	1	0	1	1	imm8

Literal variant

Applies when `Rn == 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, <label>
VLDR{<c>}{<q>}{.64} <Dd>, [PC, #{+/-}<imm>]
```

Offset variant

Applies when `Rn != 1111`.

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
CheckDecodeFaults();
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
```

T2

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	1	0	1	U	D	0	1	Rn	Vd	1	0	1	0	imm8

Literal variant

Applies when `Rn == 1111`.

```
VLDR{<c>}{<q>}{.32} <Sd>, <label>
VLDR{<c>}{<q>}{.32} <Sd>, [PC, #{+/-}<imm>]
```

Offset variant

Applies when `Rn != 1111`.

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Decode for all variants of this encoding

```
CheckDecodeFaults();
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`.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:
	`-` when `U = 0`
	`+` 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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    if single_reg then
        S[d] = MemA[address,4];
    else
        word1 = MemA[address,4];  word2 = MemA[address+4,4];
        // Combine the word-aligned words in the correct order for current endianness.
        D[d] = if BigEndian() then word1:word2 else word2:word1;
```

C2.4.270 VLLDM

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 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>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);
if !IsSecure() then UNDEFINED;
if n == 15 then UNPREDICTABLE;
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
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, FALSE, boolean UNKNOWN);
               HandleException(exc);

            for i = 0 to 15
                S[i] = MemA[R[n] + (4*i), 4];
            FPSCR = MemA[R[n] + 0x40, 4];
            if FPCCR_S.TS == '1' then
                for i = 0 to 15
                    S[i+16] = MemA[R[n] + 0x48 + (4*i), 4];
        CONTROL.FPCA = '1';
```

C2.4.271 VLSTM

Floating-point Lazy Store Multiple stores the contents of Secure floating-point registers to a prepared stack frame, and clears the Secure floating-point registers.

If floating-point lazy preservation is enabled (FPCCR.LSPEN == 1), then the next time a floating-point instruction other than VLSTM or VLLDM is executed:

- The contents of Secure floating-point registers are stored to memory.
- The Secure floating-point registers are cleared.

If Secure floating-point is not in use (CONTROL_S.SFPA == 0), this instruction behaves as a NOP.

This instruction is only available in Secure state, and is UNDEFINED in Non-secure state.

If the Floating-point Extension is not implemented, this instruction is available in Secure state, but behaves as a NOP.

T1

Armv8-M Main Extension only

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

T1 variant

```
VLSTM{<c>}{<q>} <Rn>
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
n = UInt(Rn);
if !IsSecure() then UNDEFINED;
if n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.

Operation

```
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, TRUE, TRUE);
            HandleException(exc);
```

```
else
    if !IsAligned(R[n],8) then
       UFSR.UNALIGNED = '1';
       exc = CreateException(UsageFault, FALSE, boolean UNKNOWN);
       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 FPCCR.TS == '1' then
            for i = 0 to 15
                MemA[R[n] + 0x48 + (4*i), 4] = S[i+16];
                S[i+16] = Zeros(32);
                S[i]    = Zeros(32);
            FPSCR = Zeros(32);
        else
            for i = 0 to 15
                S[i] = bits(32) UNKNOWN;
            FPSCR = bits(32) UNKNOWN;
    else
        UpdateFPCCR(R[n], FALSE);
CONTROL.FPCA = '0';
```

C2.4.272 VMAXNM

Floating-point Maximum Number determines the floating-point maximum number.

NaN handling is specified by IEEE754-2008.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 - 0	15 - 12	11	10	9	8	7	6	5	4	3 - 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	1	sz	N	0	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
maximum = (op == '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

<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
EncodingSpecificOperations();
ExecuteFPCheck();
if dp_operation then
    if maximum then
        D[d] = FPMaxNum(D[n], D[m]);
    else
        D[d] = FPMinNum(D[n], D[m]);
else
    if maximum then
        S[d] = FPMaxNum(S[n], S[m]);
    else
        S[d] = FPMinNum(S[n], S[m]);
```

C2.4.273 VMINNM

Floating-point Minimum Number determines the floating-point minimum number.

NaN handling is specified by IEEE754-2008.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	0	0	Vn	Vd	1	0	1	sz	N	1	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if InITBlock() then UNPREDICTABLE;
maximum = (op == '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

<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
EncodingSpecificOperations();
ExecuteFPCheck();
if dp_operation then
    if maximum then
        D[d] = FPMaxNum(D[n], D[m]);
    else
        D[d] = FPMinNum(D[n], D[m]);
else
    if maximum then
        S[d] = FPMaxNum(S[n], S[m]);
    else
        S[d] = FPMinNum(S[n], S[m]);
```

C2.4.274 VMLA

Floating-point Multiply Accumulate multiplies two floating-point registers, adds the product to the destination register, and places the result in the destination register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	0	0	Vn	Vd	1	0	1	sz	N	0	M	0	Vm

Single-precision scalar variant

Applies when sz == 0.

```
VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when sz == 1.

```
VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
add = (op == '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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        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);
    else
        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);
```

C2.4.275 VMLS

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.

Note

Arm recommends that software does not use the `VMLS` instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	0	0	Vn	Vd	1	0	1	sz	N	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
add = (op == '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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        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);
    else
        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);
```

C2.4.276 VMOV (between general-purpose register and single-precision register)

Floating-point Move (between general-purpose register and single-precision register) transfers the contents of a single-precision register to a general-purpose register, or the contents of a general-purpose register to a single-precision register.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn	Rt	1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

Encoding

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

Encoding

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

Decode for all variants of this encoding

```
CheckDecodeFaults();
to_arm_register = (op == '1');  t = UInt(Rt);  n = UInt(Vn:N);
if t == 15 || t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<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 *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

C2.4.277 VMOV (between two general-purpose registers and a doubleword register)

Floating-point Move (between two general-purpose registers and a doubleword register) transfers two words from two general-purpose registers to a doubleword register, or from a doubleword register to two general-purpose registers.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	1	0	0	M	1	Vm

Encoding

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

Encoding

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

Decode for all variants of this encoding

```
CheckDecodeFaults();
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
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

`<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 *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
```

```
if to_arm_registers then
    R[t] = D[m]<31:0>;
    R[t2] = D[m]<63:32>;
else
    D[m]<31:0> = R[t];
    D[m]<63:32> = R[t2];
```

C2.4.278 VMOV (between two general-purpose registers and two single-precision registers)

Floating-point Move (between two general-purpose registers and two single-precision registers) transfers the contents of two consecutively numbered single-precision registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision registers. The general-purpose registers do not have to be contiguous.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	0	0	0	1	0	op	Rt2	Rt	1	0	1	0	0	0	M	1	Vm

Encoding

Applies when `op == 1`.

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

Encoding

Applies when `op == 0`.

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

Decode for all variants of this encoding

```
CheckDecodeFaults();
to_arm_registers = (op == '1');  t = UInt(Rt);  t2 = UInt(Rt2);  m = UInt(Vm:M);
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if t == 13 || t2 == 13 then UNPREDICTABLE;
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.

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

`<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 *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.

Operation

```
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];
```

C2.4.279 VMOV (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.

Note

The pseudocode descriptions of the instruction operation convert the doubleword register description into the corresponding single-precision register. For example, D3[1], indicating the upper word of D3, becomes S7.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	H	1	Vn	Rt	1	0	1	1	N	0	0	1	(0)	(0)	(0)	(0)

T1 variant

```
VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>
```

Decode for this encoding

```
CheckDecodeFaults();
t = UInt(Rt);  n = UInt(N:Vn);
upper = (H == '1');
if t == 15 || t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<dt>`	The data size. It must be either 32 or omitted.
`<Rt>`	The destination general-purpose register, encoded in the "Rt" field.
`<Dn[x]>`	The source doubleword register and required word. The register `<Dd>` is encoded in N:Vn. `x` is 1 for the top half of the register, or 0 for the bottom half of the register, and is encoded in `H`.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if upper then
        R[t] = D[n]<63:32>;
    else
        R[t] = D[n]<31:0>;
```

C2.4.280 VMOV (immediate)

Floating-point Move (immediate) places an immediate constant into the destination floating-point register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H	Vd	1	0	1	sz	(0)	0	(0)	0	imm4L

Single-precision scalar variant

Applies when `sz == 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, #<imm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, #<imm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
if dp_operation then
    d = UInt(D:Vd);  imm64 = VFPExpandImm(imm4H:imm4L, 64);
else
    d = UInt(Vd:D);  imm32 = VFPExpandImm(imm4H:imm4L, 32);
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = imm64;
    else
        S[d] = imm32;
```

C2.4.281 VMOV (register)

Floating-point Move (immediate) copies the contents of one register to another.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	0	Vd	1	0	1	sz	0	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = D[m];
    else
        S[d] = S[m];
```

C2.4.282 VMOV (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.

Note

The pseudocode descriptions of the instruction operation convert the doubleword register description into the corresponding single-precision register. For example, D3[1], indicating the upper word of D3, becomes S7.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3			0	15			12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	H	0	Vd				Rt				1	0	1	1	D	0	0	1	(0)	(0)	(0)	(0)

T1 variant

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

Decode for this encoding

```
CheckDecodeFaults();
d = UInt(D:Vd);  t = UInt(Rt);
upper = (H == '1');
if t == 15 || t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<size>`	The data size. It must be either 32 or omitted.
`<Dd[x]>`	The destination doubleword register and required word. The register `<Dd>` is encoded in D:Vd. `x` is 1 for the top half of the register, or 0 for the bottom half of the register, and is encoded in `H`.
`<Rt>`	The source general-purpose register, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if upper then
        D[d]<63:32> = R[t];
    else
        D[d]<31:0> = R[t];
```

C2.4.283 VMRS

Move to general-purpose Register from Floating-point Special register moves the value of the FPSCR to a general-purpose register, or the values of the FPSCR condition flags to the APSR condition flags.

T1

Armv8-M Floating-point Extension only

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

T1 variant

```
VMRS{<c>}{<q>} <Rt>, FPSCR
```

Decode for this encoding

```
CheckDecodeFaults();
t = UInt(Rt);
if t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose destination register, encoded in the "Rt" field. Is one of:

`R0-R14`	General-purpose register.
`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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    SerializeVFP();
    VFPExcBarrier();
    if t == 15 then
        APSR.N = FPSCR.N;
        APSR.Z = FPSCR.Z;
        APSR.C = FPSCR.C;
        APSR.V = FPSCR.V;
    else
        R[t] = FPSCR;
```

C2.4.284 VMSR

Move to Floating-point Special register from general-purpose Register moves the value of a general-purpose register to the FPSCR.

T1

Armv8-M Floating-point Extension only

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

T1 variant

```
VMSR{<c>}{<q>} FPSCR, <Rt>
```

Decode for this encoding

```
CheckDecodeFaults();
t = UInt(Rt);
if t == 15 || t == 13 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<Rt>`	Is the general-purpose source register to be transferred to the FPSCR, encoded in the "Rt" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    SerializeVFP();
    VFPExcBarrier();
    FPSCR = R[t];
```

C2.4.285 VMUL

Floating-point Multiply multiplies two floating-point register values, and places the result in the destination floating-point register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	1	0	Vn	Vd	1	0	1	sz	N	0	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPMul(D[n], D[m], TRUE);
    else
        S[d] = FPMul(S[n], S[m], TRUE);
```

C2.4.286 VNEG

Floating-point Negate inverts the sign bit of a single-precision or double-precision register, and places the result in the destination register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd	1	0	1	sz	0	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPNeg(D[m]);
    else
        S[d] = FPNeg(S[m]);
```

C2.4.287 VNMLA

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.

Note

Arm recommends that software does not use the `VNMLA` instruction in the Round towards +Infinity and Round towards -Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	0	1	Vn	Vd	1	0	1	sz	N	1	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
type = 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        product64 = FPMul(D[n], D[m], TRUE);
        case type 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);
    else
        product32 = FPMul(S[n], S[m], TRUE);
        case type 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);
```

C2.4.288 VNMLS

Floating-point Multiply Subtract and Negate multiplies two floating-point register values, adds the negation of the floating-point value in the destination register to the product, and writes the result back to the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 … 0	15 … 12	11	10	9	8	7	6	5	4	3 … 0
1	1	1	0	1	1	1	0	0	D	0	1	Vn	Vd	1	0	1	sz	N	0	M	0	Vm
																			op			

Single-precision scalar variant

Applies when `sz == 0`.

```
VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
type = 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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        product64 = FPMul(D[n], D[m], TRUE);
        case type 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);
```

```
else
    product32 = FPMul(S[n], S[m], TRUE);
    case type 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);
```

C2.4.289 VNMUL

Floating-point Multiply and Negate multiplies two floating-point register values, and writes the negation of the result to the destination register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	1	0	Vn	Vd	1	0	1	sz	N	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VNMUL{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VNMUL{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
type = 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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        product64 = FPMul(D[n], D[m], TRUE);
        case type 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);
```

```
else
    product32 = FPMul(S[n], S[m], TRUE);
    case type 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);
```

C2.4.290 VPOP

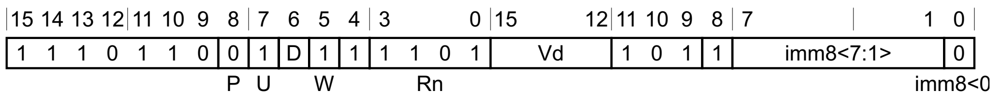
Pop FP registers from Stack loads multiple consecutive floating-point register file registers from the stack.

This instruction is an alias of the VLDM instruction. This means that:

- The encodings in this description are named to match the encodings of VLDM.
- The description of VLDM gives the operational pseudocode for this instruction.

T1

Armv8-M Floating-point Extension only



Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <dreglist>
```

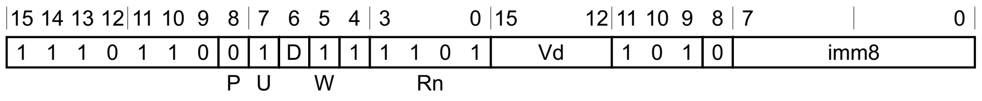
is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>
```

and is always the preferred disassembly.

T2

Armv8-M Floating-point Extension only



Increment After variant

```
VPOP{<c>}{<q>}{.<size>} <sreglist>
```

is equivalent to

```
VLDM{<c>}{<q>}{.<size>} SP!, <sreglist>
```

and is always the preferred disassembly.

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.291 VPUSH

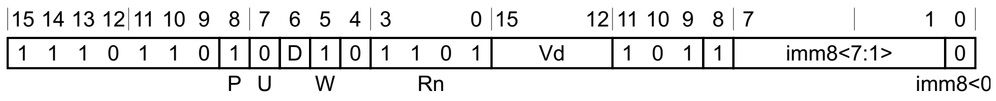
Push FP registers to Stack stores multiple consecutive registers from the floating-point register file to the stack.

This instruction is an alias of the VSTM instruction. This means that:

- The encodings in this description are named to match the encodings of VSTM.
- The description of VSTM gives the operational pseudocode for this instruction.

T1

Armv8-M Floating-point Extension only



Decrement Before variant

`VPUSH{<c>}{<q>}{.<size>} <dreglist>`

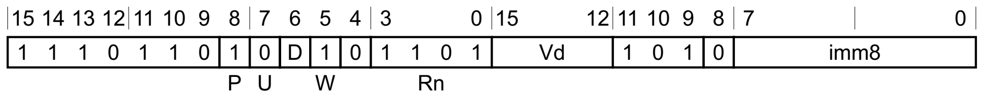
is equivalent to

`VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>`

and is always the preferred disassembly.

T2

Armv8-M Floating-point Extension only



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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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.

C2.4.292 VRINTA

Floating-point Round to Nearest Integer with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15 14 13 12 11 10 9 8 7	6	5 4 3	2	1 0	15 12	11 10 9	8	7	6	5	4	3 0
1 1 1 1 1 1 1 0 1	D	1 1 1	0	0 0	Vd	1 0 1	sz	0	1	M	0	Vm
				RM								

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTA{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTA{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

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

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;
```

```
if dp_operation then
    D[d] = FPRoundInt(D[m], rmode, away, exact);
else
    S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.293 VRINTM

Floating-point Round to Integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	1	Vd	1	0	1	sz	0	1	M	0	Vm
														RM											

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTM{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTM{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

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

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;
```

```
if dp_operation then
    D[d] = FPRoundInt(D[m], rmode, away, exact);
else
    S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.294 VRINTN

Floating-point Round to Nearest Integer with Ties to Even rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15 14 13 12 11 10 9 8 7	6	5 4 3	2	1 0	15 12	11 10 9	8	7	6	5	4	3 0
1 1 1 1 1 1 1 0 1	D	1 1 1	0	0 1	Vd	1 0 1	sz	0	1	M	0	Vm
				RM								

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTN{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTN{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

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

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;
```

```
if dp_operation then
    D[d] = FPRoundInt(D[m], rmode, away, exact);
else
    S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.295 VRINTP

Floating-point Round to Integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	0	Vd	1	0	1	sz	0	1	M	0	Vm
														RM											

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTP{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTP{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

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

`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    exact = FALSE;
```

```
if dp_operation then
    D[d] = FPRoundInt(D[m], rmode, away, exact);
else
    S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.296 VRINTR

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 the FPSCR. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	0	Vd	1	0	1	sz	0	1	M	0	Vm
																					op				

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTR{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTR{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
rmode = if op == '1' then '11' else FPSCR<23:22>;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();


    exact = FALSE;
    away  = FALSE;

    if dp_operation then
        D[d] = FPRoundInt(D[m], rmode, away, exact);
    else
        S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.297 VRINTX

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 the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15			12	11	10	9	8	7	6	5	4	3			0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	1	sz	0	1	M	0	Vm			

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTX{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTX{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();

    rmode = FPSCR<23:22>;
    away  = FALSE;
    exact = TRUE;

    if dp_operation then
```

```
        D[d] = FPRoundInt(D[m], rmode, away, exact);
    else
        S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.298 VRINTZ

Floating-point Round to Integer towards Zero rounds a floating-point value to an integral floating-point value of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	1	0	Vd	1	0	1	sz	1	1	M	0	Vm
																					op				

Single-precision scalar variant

Applies when `sz == 0`.

```
VRINTZ{<c>}{<q>}.F32.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VRINTZ{<c>}{<q>}.F64.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);

rmode = if op == '1' then '11' else FPSCR<23:22>;
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();


exact = FALSE;
    away  = FALSE;

    if dp_operation then
        D[d] = FPRoundInt(D[m], rmode, away, exact);
    else
        S[d] = FPRoundInt(S[m], rmode, away, exact);
```

C2.4.299 VSEL

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, with the effect of LT, LE, NE, and VC being achievable by exchanging the source operands.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5 4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	1	1	1	1	0	0	D	cc	Vn	Vd	1	0	1	sz	N	0	M	0	Vm

VSELEQ,doubleprec variant

Applies when `cc == 00 && sz == 1`.

```
VSELEQ.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELEQ,singleprec variant

Applies when `cc == 00 && sz == 0`.

```
VSELEQ.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELGE,doubleprec variant

Applies when `cc == 10 && sz == 1`.

```
VSELGE.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELGE,singleprec variant

Applies when `cc == 10 && sz == 0`.

```
VSELGE.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELGT,doubleprec variant

Applies when `cc == 11 && sz == 1`.

```
VSELGT.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELGT,singleprec variant

Applies when `cc == 11 && sz == 0`.

```
VSELGT.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

VSELVS,doubleprec variant

Applies when `cc == 01 && sz == 1`.

```
VSELVS.F64 <Dd>, <Dn>, <Dm> // Not permitted in IT block
```

VSELVS,singleprec variant

Applies when `cc == 01 && sz == 0`.

```
VSELVS.F32 <Sd>, <Sn>, <Sm> // Not permitted in IT block
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

<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

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
    D[d] = if ConditionHolds(cond) then D[n] else D[m];
else
    S[d] = if ConditionHolds(cond) then S[n] else S[m];
```

C2.4.300 VSQRT

Floating-point Square Root calculates the square root of a floating-point register value and writes the result to another floating-point register.

T1

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	1	D	1	1	0	0	0	1	Vd	1	0	1	sz	1	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPSqrt(D[m]);
    else
        S[d] = FPSqrt(S[m]);
```

C2.4.301 VSTM

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. See *Alias conditions* on page C2-847 for details of when each alias is preferred.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 1	0
1	1	1	0	1	1	0	P	U	D	W	0	Rn	Vd	1	0	1	1	imm8<7:1>	0
																			imm8<0

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
CheckDecodeFaults();
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = FALSE;  add = (U == '1');  wback = (W == '1');
d = UInt(D:Vd);  n = UInt(Rn);  imm32 = ZeroExtend(imm8:'00', 32);
regs = UInt(imm8) DIV 2;
if n == 15 then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if VFPSmallRegisterBank() && (d+regs) > 16 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `regs == 0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The instruction operates as a `VSTM` with the same addressing mode but stores no registers.

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.

If `VFPSmallRegisterBank() && (d+regs) > 16`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as `NOP`.
- The memory locations specified by the instruction and the number of registers specified by the instruction if the register list had not gone out of range, become UNKNOWN. If the instruction specifies writeback, then that register becomes UNKNOWN. This behavior does not affect any other memory locations.

T2

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	1	0	P	U	D	W	0	Rn	Vd	1	0	1	0	imm8

Decrement Before variant

Applies when `P == 1 && U == 0 && W == 1`.

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After variant

Applies when `P == 0 && U == 1`.

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

Decode for all variants of this encoding

```
if P == '0' && U == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE VSTR;
CheckDecodeFaults();
if P == '1' && U == '1' && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
single_regs = TRUE;  add = (U == '1');  wback = (W == '1');  d = UInt(Vd:D);  n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32);  regs = UInt(imm8);
if n == 15 then UNPREDICTABLE;
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.

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.

Notes for all encodings

Related encodings: *Floating-point load/store and 64-bit register moves* on page C2-364.

Alias conditions

Alias	is preferred when
VPUSH	`P == '1' && U == '0' && W == '1' && Rn == '1101'`

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`<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
        (limit, applylimit) = LookUpSPLim(LookUpSP());
    else
        applylimit = FALSE;

    // Memory operation only performed if limit not violated
    if !applylimit || (UInt(regval) >= UInt(limit)) 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.
                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>;
                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;
```

C2.4.302 VSTR

Floating-point Store Register stores a single Floating-point Extension register to memory, using an address from a general-purpose register, with an optional offset.

T1

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	1	0	1	U	D	0	0	Rn	Vd	1	0	1	1	imm8

T1 variant

```
VSTR{<c>}{<q>}{.64} <Dd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for this encoding

```
CheckDecodeFaults();
single_reg = FALSE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(D:Vd);  n = UInt(Rn);
if n == 15 then UNPREDICTABLE;
```

T2

Armv8-M Floating-point Extension only

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7 0
1	1	1	0	1	1	0	1	U	D	0	0	Rn	Vd	1	0	1	0	imm8

T2 variant

```
VSTR{<c>}{<q>}{.32} <Sd>, [<Rn>{, #{+/-}<imm>}]
```

Decode for this encoding

```
CheckDecodeFaults();
single_reg = TRUE;  add = (U == '1');  imm32 = ZeroExtend(imm8:'00', 32);
d = UInt(Vd:D);  n = UInt(Rn);
if n == 15 then UNPREDICTABLE;
```

Assembler symbols

`<c>`	See *Standard assembler syntax fields* on page C1-310.
`<q>`	See *Standard assembler syntax fields* on page C1-310.
`.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:

-	when `U = 0`
+	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);
    if single_reg then
        MemA[address,4] = S[d];
    else
        // Store as two word-aligned words in the correct order for current endianness.
        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>;
```

C2.4.303 VSUB

Floating-point Subtract subtracts one floating-point register value from another floating-point register value, and places the results in the destination floating-point register.

T2

Armv8-M Floating-point Extension only, sz == 1 UNDEFINED in single-precision only implementations.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15 12	11	10	9	8	7	6	5	4	3 0
1	1	1	0	1	1	1	0	0	D	1	1	Vn	Vd	1	0	1	sz	N	1	M	0	Vm

Single-precision scalar variant

Applies when `sz == 0`.

```
VSUB{<c>}{<q>}.F32 {<Sd>,} <Sn>, <Sm>
```

Double-precision scalar variant

Applies when `sz == 1`.

```
VSUB{<c>}{<q>}.F64 {<Dd>,} <Dn>, <Dm>
```

Decode for all variants of this encoding

```
dp_operation = (sz == '1');
CheckDecodeFaults(dp_operation);
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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.
<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

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ExecuteFPCheck();
    if dp_operation then
        D[d] = FPSub(D[n], D[m], TRUE);
    else
        S[d] = FPSub(S[n], S[m], TRUE);
```

C2.4.304 WFE

Wait For Event is a hint instruction. If the Event Register is clear, it suspends execution in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, exception or other event occurs.

This is a NOP-compatible hint. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

C2.4.305 WFI

Wait For Interrupt is a hint instruction. It suspends execution, in the lowest power state available consistent with a fast wakeup without the need for software restoration, until a reset, asynchronous exception or other event occurs.

This is a NOP-compatible hint. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

T1

Armv8-M

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

T1 variant

```
WFI{<c>}{<q>}
```

Decode for this encoding

```
// No additional decoding required
```

T2

Armv8-M Main Extension only

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

T2 variant

```
WFI{<c>}.W
```

Decode for this encoding

```
if !HaveMainExt() then UNDEFINED;
// No additional decoding required
```

Assembler symbols

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

C2.4.306 YIELD

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. For more information about NOP-compatible hints, see *NOP-compatible hint instructions* on page C1-319.

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

<c>	See *Standard assembler syntax fields* on page C1-310.
<q>	See *Standard assembler syntax fields* on page C1-310.

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* on page D1-859
-	*Payloads* on page D1-859
`0xE0000000`	*Instrumentation Macrocell* on page D1-859
`0xE0001000`	*Data Watchpoint and Trace* on page D1-861
`0xE0002000`	*Flash Patch and Breakpoint* on page D1-862
`0xE000E004`	*Implementation Control Block* on page D1-862
`0xE000E010`	*SysTick Timer* on page D1-862
`0xE000E100`	*Nested Vectored Interrupt Controller* on page D1-864
`0xE000ED00`	*System Control Block* on page D1-864
`0xE000ED90`	*Memory Protection Unit* on page D1-866
`0xE000EDD0`	*Security Attribution Unit* on page D1-866
`0xE000EDF0`	*Debug Control Block* on page D1-866
`0xE000EF00`	*Software Interrupt Generation* on page D1-867
`0xE000EF34`	*Floating-Point Extension* on page D1-867
`0xE000EF50`	*Cache Maintenance Operations* on page D1-867
`0xE000EFB0`	*Debug Identification Block* on page D1-867
`0xE002E004`	*Implementation Control Block (NS alias)* on page D1-869
`0xE002E010`	*SysTick Timer (NS alias)* on page D1-869
`0xE002E100`	*Nested Vectored Interrupt Controller (NS alias)* on page D1-869
`0xE002ED00`	*System Control Block (NS alias)* on page D1-869
`0xE002ED90`	*Memory Protection Unit (NS alias)* on page D1-871
`0xE002EDF0`	*Debug Control Block (NS alias)* on page D1-871
`0xE002EF00`	*Software Interrupt Generation (NS alias)* on page D1-871
`0xE002EF34`	*Floating-Point Extension (NS alias)* on page D1-871
`0xE002EF50`	*Cache Maintenance Operations (NS alias)* on page D1-872
`0xE002EFB0`	*Debug Identification Block (NS alias)* on page D1-872
`0xE0040000`	*Trace Port Interface Unit* on page D1-873

D1.1.1 Special and general-purpose registers

Name	Description
APSR	Application Program Status Register
BASEPRI	Base Priority Mask Register
CONTROL	Control Register
EPSR	Execution Program Status Register
FAULTMASK	Fault Mask Register
FPSCR	Floating-point Status and Control Register
IPSR	Interrupt Program Status Register
LR	Link Register
MSPLIM	Main Stack Pointer Limit Register
PC	Program Counter
PRIMASK	Exception Mask Register
PSPLIM	Process Stack Pointer Limit Register
Rn	General-Purpose Register *n*
SP	Current Stack Pointer Register
SP_NS	Stack Pointer (Non-secure)
XPSR	Combined Program Status Registers

D1.1.2 Payloads

Name	Description
EXC_RETURN	Exception Return Payload
FNC_RETURN	Function Return 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

Address	Register	Description
`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
`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*
`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 Implementation Control Block

Address	Register	Description
`0xE000E004`	ICTR	Interrupt Controller Type Register
`0xE000E008`	ACTLR	Auxiliary Control Register
`0xE000E00C`	CPPWR	Coprocessor Power Control Register

D1.1.7 SysTick Timer

Address	Register	Description
`0xE000E010`	SYST_CSR	SysTick Control and Status Register

Address	Register	Description
0xE000E014	SYST_RVR	SysTick Reload Value Register
0xE000E018	SYST_CVR	SysTick Current Value Register
0xE000E01C	SYST_CALIB	SysTick Calibration Value Register

D1.1.8 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*

D1.1.9 System Control Block

Address	Register	Description
`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`	CFSR	Configurable Fault Status Register
`0xE000ED28`	MMFSR	MemManage 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
`0xE000ED40`	ID_PFR0	Processor Feature Register 0
`0xE000ED44`	ID_PFR1	Processor Feature Register 1

Address	Register	Description
`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

D1.1.10 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

D1.1.11 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

D1.1.12 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
`0xE000EE04`	DAUTHCTRL	Debug Authentication Control Register
`0xE000EE08`	DSCSR	Debug Security Control and Status Register

D1.1.13 Software Interrupt Generation

Address	Register	Description
0xE000EF00	STIR	Software Triggered Interrupt Register

D1.1.14 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

D1.1.15 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

D1.1.16 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

Address	Register	Description
`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

D1.1.17 Implementation Control Block (NS alias)

Address	Register	Description
0xE002E004	ICTR_NS	Interrupt Controller Type Register (NS)
0xE002E008	ACTLR_NS	Auxiliary Control Register (NS)
0xE002E00C	CPPWR_NS	Coprocessor Power Control Register (NS)

D1.1.18 SysTick Timer (NS alias)

Address	Register	Description
0xE002E010	SYST_CSR_NS	SysTick Control and Status Register (NS)
0xE002E014	SYST_RVR_NS	SysTick Reload Value Register (NS)
0xE002E018	SYST_CVR_NS	SysTick Current Value Register (NS)
0xE002E01C	SYST_CALIB_NS	SysTick Calibration Value Register (NS)

D1.1.19 Nested Vectored Interrupt Controller (NS alias)

Address	Register	Description
0xE002E100	NVIC_ISERn_NS	Interrupt Set Enable Register *n* (NS)
0xE002E180	NVIC_ICERn_NS	Interrupt Clear Enable Register *n* (NS)
0xE002E200	NVIC_ISPRn_NS	Interrupt Set Pending Register *n* (NS)
0xE002E280	NVIC_ICPRn_NS	Interrupt Clear Pending Register *n* (NS)
0xE002E300	NVIC_IABRn_NS	Interrupt Active Bit Register *n* (NS)
0xE002E400	NVIC_IPRn_NS	Interrupt Priority Register *n* (NS)

D1.1.20 System Control Block (NS alias)

Address	Register	Description
0xE002ED00	CPUID_NS	CPUID Base Register (NS)
0xE002ED04	ICSR_NS	Interrupt Control and State Register (NS)
0xE002ED08	VTOR_NS	Vector Table Offset Register (NS)
0xE002ED0C	AIRCR_NS	Application Interrupt and Reset Control Register (NS)
0xE002ED10	SCR_NS	System Control Register (NS)
0xE002ED14	CCR_NS	Configuration and Control Register (NS)
0xE002ED18	SHPR1_NS	System Handler Priority Register 1 (NS)
0xE002ED1C	SHPR2_NS	System Handler Priority Register 2 (NS)
0xE002ED20	SHPR3_NS	System Handler Priority Register 3 (NS)

Address	Register	Description
`0xE002ED24`	SHCSR_NS	System Handler Control and State Register (NS)
`0xE002ED28`	CFSR_NS	Configurable Fault Status Register (NS)
`0xE002ED28`	MMFSR_NS	MemManage Fault Status Register (NS)
`0xE002ED29`	BFSR_NS	BusFault Status Register (NS)
`0xE002ED2A`	UFSR_NS	UsageFault Status Register (NS)
`0xE002ED2C`	HFSR_NS	HardFault Status Register (NS)
`0xE002ED30`	DFSR_NS	Debug Fault Status Register (NS)
`0xE002ED34`	MMFAR_NS	MemManage Fault Address Register (NS)
`0xE002ED38`	BFAR_NS	BusFault Address Register (NS)
`0xE002ED3C`	AFSR_NS	Auxiliary Fault Status Register (NS)
`0xE002ED40`	ID_PFR0_NS	Processor Feature Register 0 (NS)
`0xE002ED44`	ID_PFR1_NS	Processor Feature Register 1 (NS)
`0xE002ED48`	ID_DFR0_NS	Debug Feature Register 0 (NS)
`0xE002ED4C`	ID_AFR0_NS	Auxiliary Feature Register 0 (NS)
`0xE002ED50`	ID_MMFR0_NS	Memory Model Feature Register 0 (NS)
`0xE002ED54`	ID_MMFR1_NS	Memory Model Feature Register 1 (NS)
`0xE002ED58`	ID_MMFR2_NS	Memory Model Feature Register 2 (NS)
`0xE002ED5C`	ID_MMFR3_NS	Memory Model Feature Register 3 (NS)
`0xE002ED60`	ID_ISAR0_NS	Instruction Set Attribute Register 0 (NS)
`0xE002ED64`	ID_ISAR1_NS	Instruction Set Attribute Register 1 (NS)
`0xE002ED68`	ID_ISAR2_NS	Instruction Set Attribute Register 2 (NS)
`0xE002ED6C`	ID_ISAR3_NS	Instruction Set Attribute Register 3 (NS)
`0xE002ED70`	ID_ISAR4_NS	Instruction Set Attribute Register 4 (NS)
`0xE002ED74`	ID_ISAR5_NS	Instruction Set Attribute Register 5 (NS)
`0xE002ED78`	CLIDR_NS	Cache Level ID Register (NS)
`0xE002ED7C`	CTR_NS	Cache Type Register (NS)
`0xE002ED80`	CCSIDR_NS	Current Cache Size ID register (NS)
`0xE002ED84`	CSSELR_NS	Cache Size Selection Register (NS)
`0xE002ED88`	CPACR_NS	Coprocessor Access Control Register (NS)

D1.1.21 Memory Protection Unit (NS alias)

Address	Register	Description
0xE002ED90	MPU_TYPE_NS	MPU Type Register (NS)
0xE002ED94	MPU_CTRL_NS	MPU Control Register (NS)
0xE002ED98	MPU_RNR_NS	MPU Region Number Register (NS)
0xE002ED9C	MPU_RBAR_NS	MPU Region Base Address Register (NS)
0xE002EDA0	MPU_RLAR_NS	MPU Region Limit Address Register (NS)
0xE002EDA4	MPU_RBAR_An_NS	MPU Region Base Address Register Alias *n* (NS)
0xE002EDA8	MPU_RLAR_An_NS	MPU Region Limit Address Register Alias *n* (NS)
0xE002EDC0	MPU_MAIR0_NS	MPU Memory Attribute Indirection Register 0 (NS)
0xE002EDC4	MPU_MAIR1_NS	MPU Memory Attribute Indirection Register 1 (NS)

D1.1.22 Debug Control Block (NS alias)

Address	Register	Description
0xE002EDF0	DHCSR_NS	Debug Halting Control and Status Register (NS)
0xE002EDF8	DCRDR_NS	Debug Core Register Data Register (NS)
0xE002EDFC	DEMCR_NS	Debug Exception and Monitor Control Register (NS)

D1.1.23 Software Interrupt Generation (NS alias)

Address	Register	Description
0xE002EF00	STIR_NS	Software Triggered Interrupt Register (NS)

D1.1.24 Floating-Point Extension (NS alias)

Address	Register	Description
0xE002EF34	FPCCR_NS	Floating-Point Context Control Register (NS)
0xE002EF38	FPCAR_NS	Floating-Point Context Address Register (NS)
0xE002EF3C	FPDSCR_NS	Floating-Point Default Status Control Register (NS)
0xE002EF40	MVFR0_NS	Media and VFP Feature Register 0 (NS)
0xE002EF44	MVFR1_NS	Media and VFP Feature Register 1 (NS)
0xE002EF48	MVFR2_NS	Media and VFP Feature Register 2 (NS)

D1.1.25 Cache Maintenance Operations (NS alias)

Address	Register	Description
`0xE002EF50`	ICIALLU_NS	Instruction Cache Invalidate All to PoU (NS)
`0xE002EF58`	ICIMVAU_NS	Instruction Cache line Invalidate by Address to PoU (NS)
`0xE002EF5C`	DCIMVAC_NS	Data Cache line Invalidate by Address to PoC (NS)
`0xE002EF60`	DCISW_NS	Data Cache line Invalidate by Set/Way (NS)
`0xE002EF64`	DCCMVAU_NS	Data Cache line Clean by address to PoU (NS)
`0xE002EF68`	DCCMVAC_NS	Data Cache line Clean by Address to PoC (NS)
`0xE002EF6C`	DCCSW_NS	Data Cache Clean line by Set/Way (NS)
`0xE002EF70`	DCCIMVAC_NS	Data Cache line Clean and Invalidate by Address to PoC (NS)
`0xE002EF74`	DCCISW_NS	Data Cache line Clean and Invalidate by Set/Way (NS)
`0xE002EF78`	BPIALL_NS	Branch Predictor Invalidate All (NS)

D1.1.26 Debug Identification Block (NS alias)

Address	Register	Description
`0xE002EFB0`	DLAR_NS	SCS Software Lock Access Register (NS)
`0xE002EFB4`	DLSR_NS	SCS Software Lock Status Register (NS)
`0xE002EFB8`	DAUTHSTATUS_NS	Debug Authentication Status Register (NS)
`0xE002EFBC`	DDEVARCH_NS	SCS Device Architecture Register (NS)
`0xE002EFCC`	DDEVTYPE_NS	SCS Device Type Register (NS)
`0xE002EFD0`	DPIDR4_NS	SCS Peripheral Identification Register 4 (NS)
`0xE002EFD4`	DPIDR5_NS	SCS Peripheral Identification Register 5 (NS)
`0xE002EFD8`	DPIDR6_NS	SCS Peripheral Identification Register 6 (NS)
`0xE002EFDC`	DPIDR7_NS	SCS Peripheral Identification Register 7 (NS)
`0xE002EFE0`	DPIDR0_NS	SCS Peripheral Identification Register 0 (NS)
`0xE002EFE4`	DPIDR1_NS	SCS Peripheral Identification Register 1 (NS)
`0xE002EFE8`	DPIDR2_NS	SCS Peripheral Identification Register 2 (NS)
`0xE002EFEC`	DPIDR3_NS	SCS Peripheral Identification Register 3 (NS)
`0xE002EFF0`	DCIDR0_NS	SCS Component Identification Register 0 (NS)
`0xE002EFF4`	DCIDR1_NS	SCS Component Identification Register 1 (NS)
`0xE002EFF8`	DCIDR2_NS	SCS Component Identification Register 2 (NS)
`0xE002EFFC`	DCIDR3_NS	SCS Component Identification Register 3 (NS)

D1.1.27 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
`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

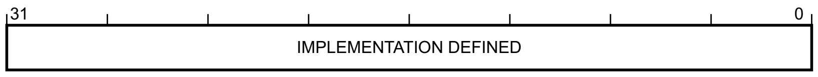
D1.2.1 ACTLR, Auxiliary Control Register

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.

Field descriptions

The ACTLR bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED. The contents of this field are IMPLEMENTATION DEFINED.

D1.2.2 AFSR, Auxiliary Fault Status Register

The AFSR characteristics are:

Purpose Provides IMPLEMENTATION DEFINED fault status information.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

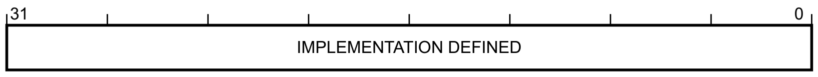
Attributes 32-bit read/write register located at `0xE000ED3C`.

Secure software can access the Non-secure version of this register via AFSR_NS located at `0xE002ED3C`. The location `0xE002ED3C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The AFSR bit assignments are:



IMPLEMENTATION DEFINED, bits [31:0]

IMPLEMENTATION DEFINED. The contents of this field are IMPLEMENTATION DEFINED.

D1.2.3 AIRCR, Application Interrupt and Reset Control Register

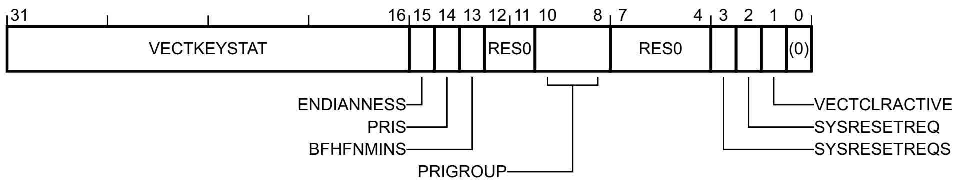
The AIRCR characteristics are:

Purpose	Sets or returns interrupt control and reset configuration.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000ED0C`. Secure software can access the Non-secure version of this register via AIRCR_NS located at `0xE002ED0C`. The location `0xE002ED0C` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states on a bit by bit basis.

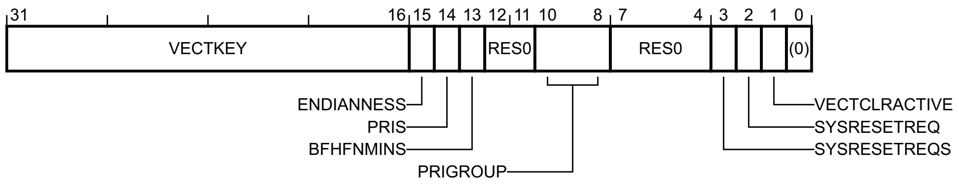
Field descriptions

The AIRCR bit assignments are:

On a read:



On a write:



VECTKEY, bits [31:16], on a write

Vector key. Writes to the AIRCR must be accompanied by a write of the value `0x05FA` to this field. Writes to the AIRCR fields that are not accompanied by this value are ignored for the purpose of updating any of the AIRCR values or initiating any AIRCR functionality.

This field is not banked between Security states.

The possible values of this field are:

`0x05FA`	Permit write to AIRCR fields.
Not `0x05FA`	Accompanying write to AIRCR fields ignored.

VECTKEYSTAT, bits [31:16], on a read

Vector key status. Returns the bitwise inverse of the value required to be written to VECTKEY.

This field is not banked between Security states.

This field reads as `0xFA05`.

ENDIANNESS, bit [15]

Data endianness. Indicates how the PE interprets the memory system data endianness.

This bit is not banked between Security states.

The possible values of this bit are:

0	Little-endian.
1	Big-endian.

This bit is read-only.

This bit reads as an IMPLEMENTATION DEFINED value.

PRIS, bit [14]

Prioritize Secure exceptions. The value of this bit defines whether Secure exception priority boosting is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

0	Priority ranges of Secure and Non-secure exceptions are identical.
1	Non-secure exceptions are de-prioritized.

To allow lock down of this bit, it is IMPLEMENTATION DEFINED whether this bit is writable.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

BFHFNMINS, bit [13]

BusFault, HardFault, and NMI Non-secure enable. The value of this bit defines whether BusFault and NMI exceptions are Non-secure, and whether exceptions target the Non-secure HardFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

0	BusFault, HardFault, and NMI are Secure.
1	BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault.

If an implementation resets into Secure state, this bit resets to zero. If an implementation does not support Secure state, this bit is RAO/WI. To allow lock down of this field it is IMPLEMENTATION DEFINED whether this bit is writable. The effect of setting both BFHFNMINS and PRIS to 1 is UNPREDICTABLE.

This bit is read-only from Non-secure state.

This bit resets to zero on a Warm reset.

Bits [12:11]

Reserved, RES0.

PRIGROUP, bits [10:8]

Priority grouping. The value of this field defines the exception priority binary point position for the selected Security state.

This field is banked between Security states.

The possible values of this field are:

`0b000`	Group priority [7:1], subpriority [0].
`0b001`	Group priority [7:2], subpriority [1:0].
`0b010`	Group priority [7:3], subpriority [2:0].
`0b011`	Group priority [7:4], subpriority [3:0].
`0b100`	Group priority [7:5], subpriority [4:0].
`0b101`	Group priority [7:6], subpriority [5:0].
`0b110`	Group priority [7], subpriority [6:0].
`0b111`	No group priority, subpriority [7:0].

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

Bits [7:4]

Reserved, RES0.

SYSRESETREQS, bit [3]

System reset request Secure only. The value of this bit defines whether the SYSRESETREQ bit is functional for Non-secure use.

This bit is not banked between Security states.

The possible values of this bit are:

0	SYSRESETREQ functionality is available to both Security states.
1	SYSRESETREQ functionality is only available to Secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

SYSRESETREQ, bit [2]

System reset request. This bit allows software or a debugger to request a system reset.

This bit is not banked between Security states.

The possible values of this bit are:

0	Do not request a system reset.
1	Request a system reset.

When SYSRESETREQS is set to 1, the Non-secure view of this bit is RAZ/WI.

This bit resets to zero on a Warm reset.

VECTCLRACTIVE, bit [1]

Clear active state.

A debugger write of one to this bit when the PE is halted in Debug state:

- IPSR is cleared to zero.
- The active state for all Non-secure exceptions is cleared.
- If DHCSR.S_SDE==1, the active state for all Secure exceptions is cleared.

This bit is not banked between Security states.

The possible values of this bit are:

0	Do not clear active state.
1	Clear active state.

Writes to this bit while the PE is in Non-debug state are ignored.

This bit reads as zero.

Bit [0]

Reserved, RES0.

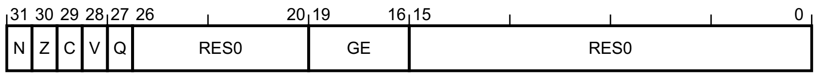
D1.2.4 APSR, Application Program Status Register

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.

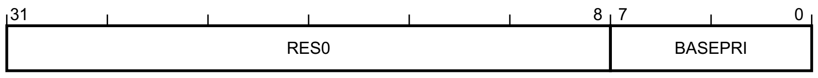
D1.2.5 BASEPRI, Base Priority Mask Register

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:

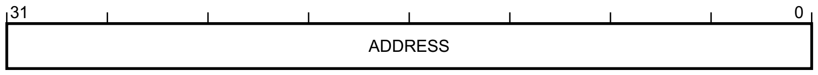
Purpose	Shows the address associated with a precise data access BusFault.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension 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.

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. The BFSR shows the reason for the fault. This field is valid only when BFSR.BFARVALID is set, otherwise it is UNKNOWN.

In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if MMFSR.MMARVALID is set.

If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

This field resets to an UNKNOWN value on a Warm reset.

——— **Note** ———

If an implementation shares a common BFAR and MMFAR it must not leak Secure state information to the Non-secure state. One possible implementation is that BFAR shares resource with the Secure MMFAR if AIRCR.BFHFNMINS is zero, and with the Non-secure MMFAR if AIRCR.BFHFNMINS is set.

———————————

D1.2.7 BFSR, BusFault Status Register

The BFSR characteristics are:

Purpose Shows the status of bus errors resulting from instruction fetches and data accesses.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Attributes 8-bit read/write-one-to-clear register located at `0xE000ED29`.

Secure software can access the Non-secure version of this register via BFSR_NS located at `0xE002ED29`. The location `0xE002ED29` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

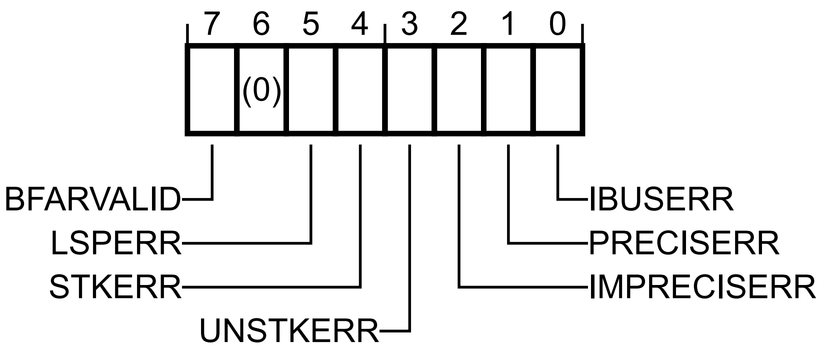
This register is part of CFSR.

Preface

The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is set to 0.

Field descriptions

The BFSR bit assignments are:



BFARVALID, bit [7]

BFAR valid. Indicates validity of the contents of the BFAR register.

The possible values of this bit are:

0	BFAR content not valid.
1	BFAR content valid.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

Bit [6]

Reserved, RES0.

LSPERR, bit [5]

Lazy state preservation error. Records whether a 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.

Configurations This register is always implemented.

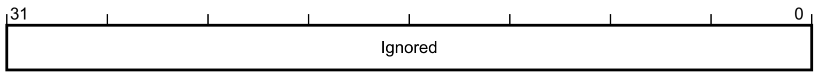
Attributes 32-bit write-only register located at `0xE000EF78`.

Secure software can access the Non-secure version of this register via BPIALL_NS located at `0xE002EF78`. The location `0xE002EF78` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The BPIALL bit assignments are:



Ignored, bits [31:0]

Ignored. The value written to this field is ignored.

D1.2.9 CCR, Configuration and Control Register

The CCR characteristics are:

Purpose Sets or returns configuration and control data.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

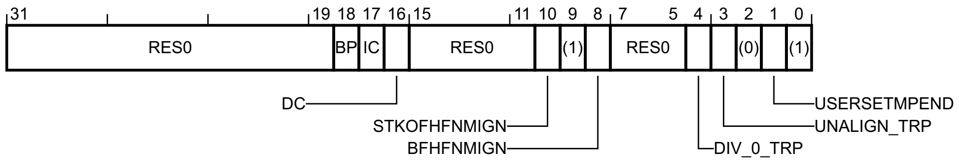
Attributes 32-bit read/write register located at `0xE000ED14`.

Secure software can access the Non-secure version of this register via CCR_NS located at `0xE002ED14`. The location `0xE002ED14` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

Field descriptions

The CCR bit assignments are:



Bits [31:19]

Reserved, RES0.

BP, bit [18]

Branch prediction enable. Enables program flow prediction for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

0	Program flow prediction disabled for the selected Security state.
1	Program flow prediction enabled for the selected Security state.

If program flow prediction cannot be disabled, this bit is RAO/WI. If the program flow prediction is not supported, this bit is RAZ/WI.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

IC, bit [17]

Instruction cache enable. This is a global enable bit for instruction caches in the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

0	Instruction caches disabled for the selected Security state.
1	Instruction caches enabled for the selected Security state.

If the PE does not implement instruction caches, this bit is RAZ/WI.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

DC, bit [16]

Data cache enable. Enables data caching of all data accesses to Normal memory.

This bit is banked between Security states.

The possible values of this bit are:

0	Data caching disabled.
1	Data caching enabled.

The secure version of this bit controls the Cacheability of accesses to secure memory.

The non-secure version of this bit controls the Cacheability of accesses to non-secure memory.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

Bits [15:11]

Reserved, RES0.

STKOFHFNMIGN, bit [10]

Stack overflow in HardFault and NMI ignore. Controls the effect of a stack limit violation while executing at a requested priority less than 0 for the Security state with which the stack limit register is associated.

This bit is banked between Security states.

The possible values of this bit are:

0	Stack limit faults not ignored.
1	Stack limit faults at requested priorities of less than 0 ignored.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

Bit [9]

Reserved, RES1.

BFHFNMIGN, bit [8]

BusFault in HardFault or NMI ignore. Determines the effect of precise BusFaults on handlers running at a requested priority less than 0.

This bit is not banked between Security states.

The possible values of this bit are:

0	Precise BusFaults not ignored.
1	Precise BusFaults at requested priorities of less than 0 ignored.

If AIRCR.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:

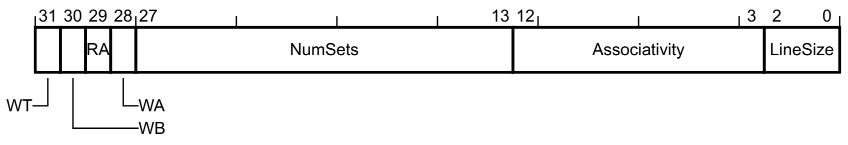
Purpose	The CCSIDR provides information about the architecture of the currently selected cache.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If CSSELR points to an unimplemented 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.

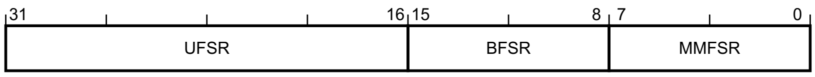
D1.2.11 CFSR, Configurable Fault Status Register

The CFSR characteristics are:

Purpose	Contains the three Configurable Fault Status Registers.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. If the Main Extension is implemented, this register is word, halfword, and byte accessible. If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write-one-to-clear register located at `0xE000ED28`. Secure software can access the Non-secure version of this register via CFSR_NS located at `0xE002ED28`. The location `0xE002ED28` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states on a bit by bit basis.

Field descriptions

The CFSR bit assignments are:



UFSR, bits [31:16]

UsageFault Status Register. Provides information on UsageFault exceptions.

This field is banked between Security states.

See UFSR.

This field resets to zero on a Warm reset.

BFSR, bits [15:8]

BusFault Status Register. Provides information on BusFault exceptions.

This field is not banked between Security states.

See BFSR.

This field resets to zero on a Warm reset.

MMFSR, bits [7:0]

MemManage Fault Status Register. Provides information on MemManage exceptions.

This field is banked between Security states.

See MMFSR.

This field resets to zero on a Warm reset.

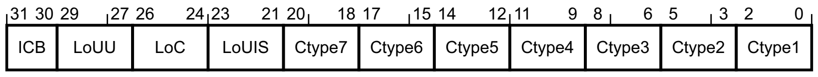
D1.2.12 CLIDR, Cache Level ID Register

The CLIDR characteristics are:

Purpose	Identifies the type of caches implemented and the level of coherency and unification.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000ED78`. Secure software can access the Non-secure version of this register via CLIDR_NS located at `0xE002ED78`. The location `0xE002ED78` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The CLIDR bit assignments are:



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.

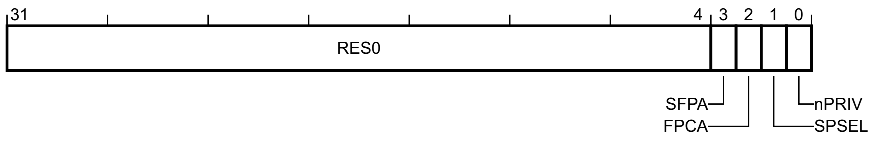
D1.2.13 CONTROL, Control Register

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 is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

FPCA, bit [2]

Floating-point context active. Defines whether the FP Extension is active in the current context.

This bit is not banked between Security states.

The possible values of this bit are:

0	FP Extension is not active.
1	FP Extension is active.

When NSACR.CP10 is set to zero, the Non-secure view of this bit is read-only. If FPCCR.ASPEN is set to 1, enabling automatic floating-point state preservation, then the PE sets this bit to 1 on successful completion of any floating-point instruction.

If the Floating-point Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

SPSEL, bit [1]

Stack-pointer select. Defines the stack pointer to be used.

This bit is banked between Security states.

The possible values of this bit are:

0	Use SP_main as the current stack.
1	In Thread mode use SP_process as the current stack.

This bit resets to zero on a Warm reset.

nPRIV, bit [0]

Not privileged. Defines the execution privilege in Thread mode.

This bit is banked between Security states.

The possible values of this bit are:

0	Thread mode has privileged access.
1	Thread mode has unprivileged access only.

If the Main Extension is not implemented, it is IMPLEMENTATION DEFINED whether this field is RW or RAZ/WI.

This bit resets to zero on a Warm reset.

D1.2.14 CPACR, Coprocessor Access Control Register

The CPACR characteristics are:

Purpose Specifies the access privileges for coprocessors and the Floating-point Extension.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Attributes 32-bit read/write register located at `0xE000ED88`.

Secure software can access the Non-secure version of this register via CPACR_NS located at `0xE002ED88`. The location `0xE002ED88` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

Field descriptions

The CPACR bit assignments are:

31 24	23 22	21 20	19 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
RES0	CP11	CP10	RES0	CP7	CP6	CP5	CP4	CP3	CP2	CP1	CP0

Bits [31:24]

Reserved, RES0.

CP11, bits [23:22]

CP11 Privilege. The value in this field is ignored. If the implementation does not include the FP Extension, this field is RAZ/WI. If the value of this bit is not programmed to the same value as the CP10 field, then the value is UNKNOWN.

This field resets to an UNKNOWN value on a Warm reset.

CP10, bits [21:20]

CP10 Privilege. Defines the access rights for the floating-point functionality.

The possible values of this field are:

`0b00`	All accesses to the FP Extension result in NOCP UsageFault.
`0b01`	Unprivileged accesses to the FP Extension result in NOCP UsageFault.
`0b11`	Full access to the FP Extension.

All other values are reserved.

The features controlled by this field are:

- The execution of any instructions within the encoding space defined by IsCPInstruction().
- Access to any floating-point registers in the range D0-D16.

If the implementation does not include the Floating-point Extension, this field is RAZ/WI. See individual floating-point instruction pages for details.

This field resets to an UNKNOWN value on a Warm reset.

Bits [19:16]

Reserved, RES0.

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.

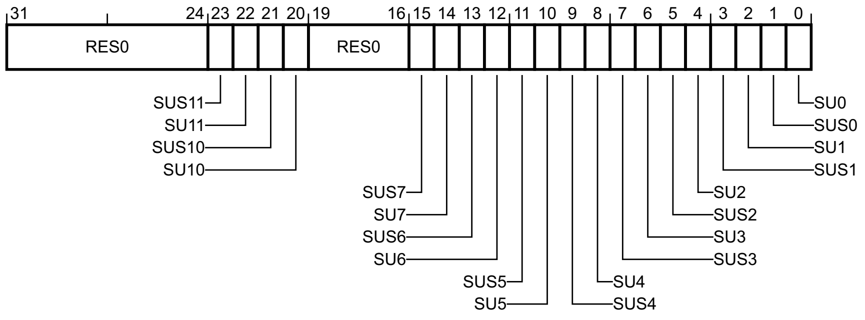
Attributes 32-bit read/write register located at `0xE000E00C`.

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.

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 SUS11 is RAZ/WI this field is also RAZ/WI. If the value of this bit is not programmed to the same value as the SUS10 field, then the value is UNKNOWN.

This bit resets to zero on a Warm reset.

SU11, bit [22]

State UNKNOWN 11. The value in this field is ignored. If SUS11 is RAZ/WI this field is also RAZ/WI. If the value of this bit is not programmed to the same value as the SU10 field, then the value is UNKNOWN.

This bit resets to zero on a Warm reset.

SUS10, bit [21]

State UNKNOWN Secure only 10. This bit indicates and allows modification of whether the SU10 field can be modified from Non-secure state.

The possible values of this bit are:

0	The SU10 field is accessible from both Security states.
1	The SU10 field is only accessible from the Secure state.

If SU10 is always RAZ/WI this field is also RAZ/WI.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

SU10, bit [20]

State UNKNOWN 10. This bit indicates and allows modification of whether the state associated with the floating-point unit is permitted to become UNKNOWN. This can be used as a hint to power control logic that the floating-point unit might be powered down.

The possible values of this bit are:

0	The floating-point state is not permitted to become UNKNOWN.
1	The floating-point state is permitted to become UNKNOWN.

When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. It is IMPLEMENTATION DEFINED whether this bit is always RAZ/WI.

This bit resets to zero on a Warm reset.

Bits [19:16]

Reserved, RES0.

SUS*m*, bit [2*m*+1], for *m* = 0 to 7

State UNKNOWN Secure only *m*. This field indicates and allows modification of whether the SU*m* field can be modified from Non-secure state.

The possible values of this field are:

0	The SU*m* field is accessible from both Security states.
1	The SU*m* field is only accessible from the Secure state.

If SU*m* is always RAZ/WI this field is also RAZ/WI.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

SU*m*, bit [2*m*], for *m* = 0 to 7

State UNKNOWN *m*. This field indicates and allows modification of whether the state associated with coprocessor *m* is permitted to become UNKNOWN. This can be used as a hint to power control logic that the coprocessor might be powered down.

The possible values of this field are:

0	The coprocessor state is not permitted to become UNKNOWN.
1	The coprocessor state is permitted to become UNKNOWN.

When 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.

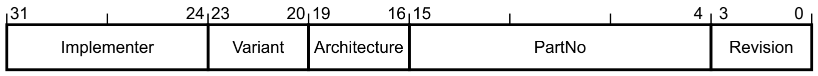
D1.2.16 CPUID, CPUID Base Register

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.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000ED00`. Secure software can access the Non-secure version of this register via CPUID_NS located at `0xE002ED00`. The location `0xE002ED00` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The CPUID bit assignments are:



Implementer, bits [31:24]

Implementer code. This field must hold an implementer code that has been assigned by Arm.

The possible values of this field are:

`0x41`	'A': Arm Limited.
Not `0x41`	Implementer other than Arm Limited.

Arm can assign codes that are not published in this manual. All values not assigned by Arm are reserved and must not be used.

This field reads as an IMPLEMENTATION DEFINED value.

Variant, bits [23:20]

Variant number. IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish between different product variants, or major revisions of a product.

This field reads as an IMPLEMENTATION DEFINED value.

Architecture, bits [19:16]

Architecture version. Defines the Architecture implemented by the PE.

The possible values of this field are:

`0b1100`	Armv8-M architecture without Main Extension.
`0b1111`	Armv8-M architecture with Main Extension.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

PartNo, bits [15:4]

Part number. IMPLEMENTATION DEFINED primary part number for the device.

This field reads as an IMPLEMENTATION DEFINED value.

Revision, bits [3:0]

Revision number. IMPLEMENTATION DEFINED revision number for the device.

This field reads as an IMPLEMENTATION DEFINED value.

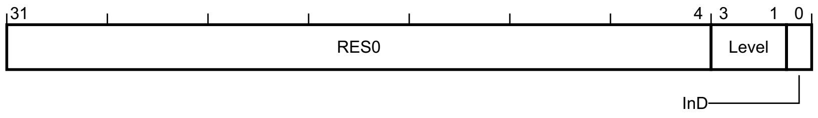
D1.2.17 CSSELR, Cache Size Selection Register

The CSSELR characteristics are:

Purpose	Selects the current Cache Size ID Register, CCSIDR, by specifying the required cache level and the cache type (either instruction or data cache)
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000ED84`. Secure software can access the Non-secure version of this register via CSSELR_NS located at `0xE002ED84`. The location `0xE002ED84` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Field descriptions

The CSSELR bit assignments are:



Bits [31:4]

Reserved, RES0.

Level, bits [3:1]

Cache level. Selects which cache level is to be identified. Permitted values are from `0b000`, indicating Level 1 cache, to `0b110` indicating Level 7 cache.

The possible values of this field are:

`0b000`	Level 1 cache.
`0b001`	Level 2 cache.
`0b010`	Level 3 cache.
`0b011`	Level 4 cache.
`0b100`	Level 5 cache.
`0b101`	Level 6 cache.
`0b110`	Level 7 cache.

All other values are reserved.

This field resets to an UNKNOWN value on a Warm reset.

InD, bit [0]

Instruction not data. Selects whether the instruction or the data cache is to be identified.

The possible values of this bit are:

0	Data or unified cache.
1	Instruction cache.

This bit resets to an UNKNOWN value on a Warm reset.

D1.2.18 CTR, Cache Type Register

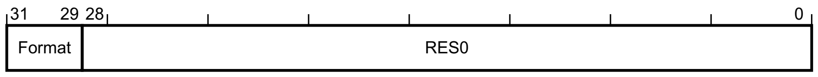
The CTR characteristics are:

Purpose	Provides information about the architecture of the caches.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000ED7C`. Secure software can access the Non-secure version of this register via CTR_NS located at `0xE002ED7C`. The location `0xE002ED7C` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

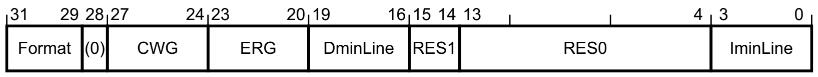
Field descriptions

The CTR bit assignments are:

When `Format!='0b100'`:



When `Format=='0b100'`:



Format, bits [31:29]

Cache Type Register format. Indicates whether cache type information is provided.

The possible values of this field are:

`0b000`	No cache type information is provided.
`0b100`	Cache type information is provided.

All other values are reserved.

The value of CLIDR is an IMPLEMENTATION DEFINED choice between `0b000` and `0b100`.

If CLIDR is nonzero then this field must read as `0b100`.

Bits [28:0], when Format!='`0b100`'

Reserved, RES0.

Bit [28], when Format=='`0b100`'

Reserved, RES0.

CWG, bits [27:24], when Format=='`0b100`'

Cache Write-Back Granule. Log_2 of the number of words of the maximum size of memory that can be overwritten as a result of the eviction of a cache entry that has had a memory location in it modified.

The possible values of this field are:

`0b0000`	Indicates that this register does not provide Cache Write-Back Granule information and either the architectural maximum of 512 words (2KB) must be assumed, or the Cache Write-Back Granule can be determined from maximum cache line size encoded in the Cache Size ID Registers.

`0b0001-0b1000`	Log_2 of the number of words.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

ERG, bits [23:20], when Format == '`0b100`'

Exclusives Reservation Granule. Log_2 of the number of words of the maximum size of the reservation granule that has been implemented for the Load-Exclusive and Store-Exclusive instructions.

The possible values of this field are:

`0b0000`	Indicates that this register does not provide Exclusives Reservation Granule information and the architectural maximum of 512 words (2KB) must be assumed.
`0b0001-0b1000`	Log_2 of the number of words.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

DminLine, bits [19:16], when Format == '`0b100`'

Data cache minimum line length. Log_2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the PE.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [15:14], when Format == '`0b100`'

Reserved, RES1.

Bits [13:4], when Format == '`0b100`'

Reserved, RES0.

IminLine, bits [3:0], when Format == '`0b100`'

Instruction cache minimum line length. Log_2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the PE.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.19 DAUTHCTRL, Debug Authentication Control Register

The DAUTHCTRL characteristics are:

Purpose This register allows the IMPLEMENTATION DEFINED authentication interface to be overridden from software.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RES0 if accessed via the debugger.

Configurations Present only if Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

Present only if the Security Extension is implemented.

This register is RES0 if the Security Extension is not implemented.

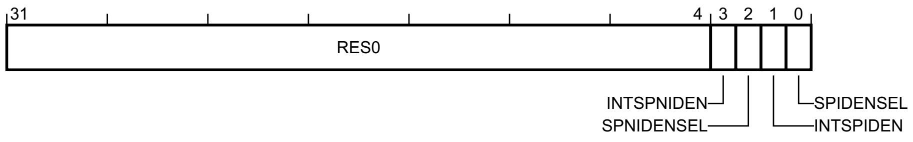
Attributes 32-bit read/write register located at `0xE000EE04`.

This register is RAZ/WI when accessed as Non-secure.

This register is not banked between Security states.

Field descriptions

The DAUTHCTRL bit assignments are:



Bits [31:4]

Reserved, RES0.

INTSPNIDEN, bit [3]

Internal Secure non-invasive debug enable. Overrides the external Secure non-invasive debug authentication interface.

The possible values of this bit are:

0	Secure Non-invasive debug prohibited.
1	Secure Non-invasive debug allowed.

Ignored if DAUTHCTRL.SPNIDENSEL == 0. See `SecureNoninvasiveDebugAllowed()`.

This bit resets to zero on a Cold reset.

SPNIDENSEL, bit [2]

Secure non-invasive debug enable select. Selects between DAUTHCTRL and the IMPLEMENTATION DEFINED external authentication interface for control of Secure non-invasive debug.

The possible values of this bit are:

0	Secure non-invasive debug controlled by the IMPLEMENTATION DEFINED external authentication interface. In the CoreSight authentication interface, this is controlled by the SPNIDEN signal.
1	Secure non-invasive debug controlled by DAUTHCTRL.INTSPNIDEN.

The PE ignores the value of this bit and Secure non-invasive debug is allowed if DHCSR.S_SDE == 1. See `SecureNoninvasiveDebugAllowed()`.

This bit resets to zero on a Cold reset.

INTSPIDEN, bit [1]

Internal Secure invasive debug enable. Overrides the external Secure invasive debug authentication interfaces.

The possible values of this bit are:

0	Secure halting and self-hosted debug prohibited.
1	Secure halting and self-hosted debug allowed.

Ignored if DAUTHCTRL.SPIDENSEL == 0. See `SecureHaltingDebugAllowed()` and `SecureDebugMonitorAllowed()`.

This bit resets to zero on a Cold reset.

SPIDENSEL, bit [0]

Secure invasive debug enable select. Selects between DAUTHCTRL and the IMPLEMENTATION DEFINED external authentication interface for control of Secure invasive debug.

The possible values of this bit are:

0	Secure halting and self-hosted debug controlled by the IMPLEMENTATION DEFINED external authentication interface. In the CoreSight authentication interface, both are controlled by the SPIDEN signal.
1	Secure halting and self-hosted debug controlled by DAUTHCTRL.INTSPIDEN.

See `SecureHaltingDebugAllowed()` and `SecureDebugMonitorAllowed()`.

This bit resets to zero on a Cold reset.

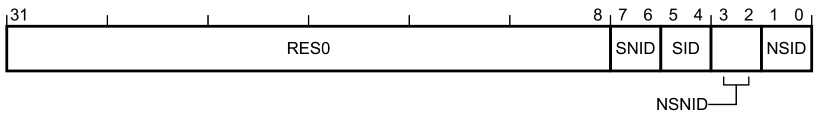
D1.2.20 DAUTHSTATUS, Debug Authentication Status Register

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. 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 [31:8]

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.

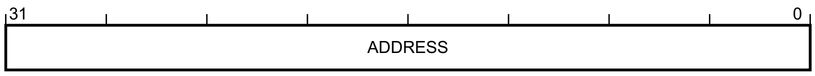
D1.2.21 DCCIMVAC, Data Cache line Clean and Invalidate by Address to PoC

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.
Configurations	This register is always implemented.
Attributes	32-bit write-only register located at `0xE000EF70`. Secure software can access the Non-secure version of this register via DCCIMVAC_NS located at `0xE002EF70`. The location `0xE002EF70` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The DCCIMVAC bit assignments are:



ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

D1.2.22 DCCISW, Data Cache line Clean and Invalidate by Set/Way

The DCCISW characteristics are:

Purpose Clean and invalidate data or unified cache line by set/way.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

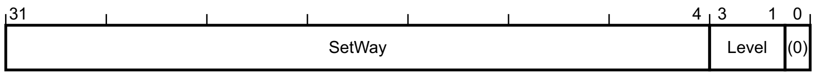
Attributes 32-bit write-only register located at `0xE000EF74`.

Secure software can access the Non-secure version of this register via DCCISW_NS located at `0xE002EF74`. The location `0xE002EF74` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DCCISW bit assignments are:



SetWay, bits [31:4]

Cache set/way. Contains two fields: Way, bits[31:32-A], the number of the way to operate on. Set, bits[B-1:L], the number of the set to operate on. Bits[L-1:4] are RES0. A = Log_2(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.

Configurations This register is always implemented.

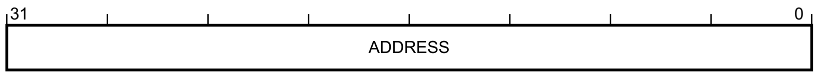
Attributes 32-bit write-only register located at `0xE000EF68`.

Secure software can access the Non-secure version of this register via DCCMVAC_NS located at `0xE002EF68`. The location `0xE002EF68` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DCCMVAC bit assignments are:



ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

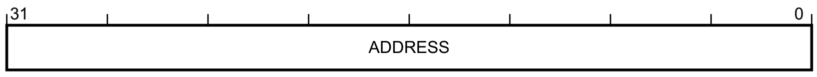
D1.2.24 DCCMVAU, Data Cache line Clean by address to PoU

The DCCMVAU characteristics are:

Purpose	Clean data or unified cache line by address to PoU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit write-only register located at `0xE000EF64`. Secure software can access the Non-secure version of this register via DCCMVAU_NS located at `0xE002EF64`. The location `0xE002EF64` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The DCCMVAU bit assignments are:



ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

D1.2.25 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.

Configurations This register is always implemented.

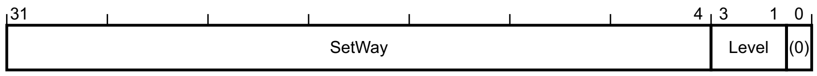
Attributes 32-bit write-only register located at `0xE000EF6C`.

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 discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

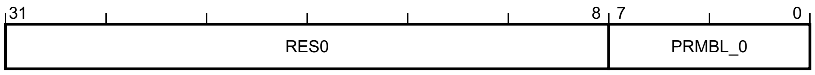
Attributes 32-bit read-only register located at `0xE000EFF0`.

Secure software can access the Non-secure version of this register via DCIDR0_NS located at `0xE002EFF0`. The location `0xE002EFF0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DCIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

D1.2.27 DCIDR1, SCS Component Identification Register 1

The DCIDR1 characteristics are:

Purpose Provides CoreSight discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

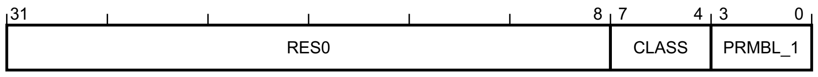
Attributes 32-bit read-only register located at `0xE000EFF4`.

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 discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

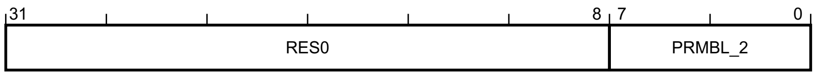
Attributes 32-bit read-only register located at `0xE000EFF8`.

Secure software can access the Non-secure version of this register via DCIDR2_NS located at `0xE002EFF8`. The location `0xE002EFF8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DCIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

D1.2.29 DCIDR3, SCS Component Identification Register 3

The DCIDR3 characteristics are:

Purpose Provides CoreSight discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

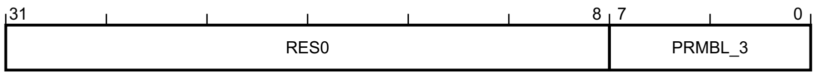
Attributes 32-bit read-only register located at `0xE000EFFC`.

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`.

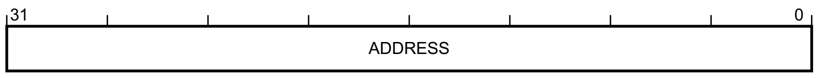
D1.2.30 DCIMVAC, Data Cache line Invalidate by Address to PoC

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.

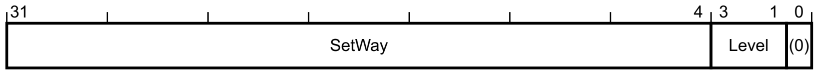
D1.2.31 DCISW, Data Cache line Invalidate by Set/Way

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.

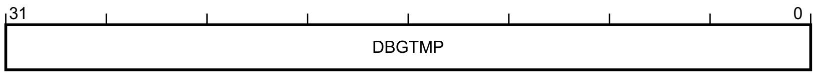
D1.2.32 DCRDR, Debug Core Register Data Register

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.
	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.

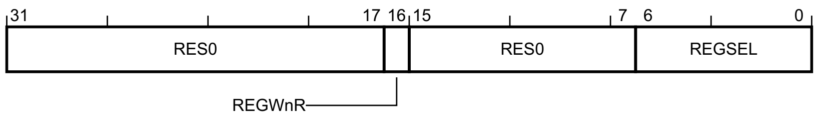
D1.2.33 DCRSR, Debug Core Register Select Register

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. Writes to this register while the PE is in Non-debug state are ignored. This register is accessible only to the debugger and RES0 to software.
Configurations	Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit write-only register located at `0xE000EDF4`. This register is not banked between Security states.

Field descriptions

The DCRSR bit assignments are:



Bits [31:17]

Reserved, RES0.

REGWnR, bit [16]

Register write/not-read. Specifies the access type for the transfer.

The possible values of this bit are:

0	Read.
1	Write.

Bits [15:7]

Reserved, RES0.

REGSEL, bits [6:0]

Register selector. Specifies the general-purpose register, special-purpose register, or Floating-point Extension register to transfer.

The possible values of this field are:

`0b0000000-0b0001100`	General-purpose registers R0-R12.
`0b0001101`	Current stack pointer, SP.
`0b0001110`	LR.
`0b0001111`	DebugReturnAddress.
`0b0010000`	XPSR.
`0b0010001`	Current state main stack pointer, SP_main.
`0b0010010`	Current state process stack pointer, SP_process.
`0b0010100`	Current state {CONTROL[7:0],FAULTMASK[7:0],BASEPRI[7:0],PRIMASK[7:0]}.

If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0.

`0b0011000` Non-secure main stack pointer, MSP_NS.
If the Security Extension is not implemented, this value is reserved.

`0b0011001` Non-secure process stack pointer, PSP_NS.
If the Security Extension is not implemented, this value is reserved.

`0b0011010` Secure main stack pointer, MSP_S. Accessible only when DHCSR.S_SDE == 1.
If the Security Extension is not implemented, this value is reserved.

`0b0011011` Secure process stack pointer, PSP_S. Accessible only when DHCSR.S_SDE == 1.
If the Security Extension is not implemented, this value is reserved.

`0b0011100` Secure main stack limit, MSPLIM_S. Accessible only when DHCSR.S_SDE == 1.
If the Security Extension is not implemented, this value is reserved.

`0b0011101` Secure process stack limit, PSPLIM_S. Accessible only when DHCSR.S_SDE == 1.
If the Security Extension is not implemented, this value is reserved.

`0b0011110` Non-secure main stack limit, MSPLIM_NS.
If the Main Extension is not implemented, this value is reserved.

`0b0011111` Non-secure process stack limit, PSPLIM_NS.
If the Main Extension is not implemented, this value is reserved.

`0b0100001` FPSCR.
If the Floating-point Extension is not implemented, this value is reserved.

`0b0100010` {CONTROL_S[7:0],FAULTMASK_S[7:0],BASEPRI_S[7:0],PRIMASK_S[7:0]}. Accessible only when DHCSR.S_SDE == 1.
If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0. If the Security Extension is not implemented, this value is reserved.

`0b0100011`
{CONTROL_NS[7:0],FAULTMASK_NS[7:0],BASEPRI_NS[7:0],PRIMASK_NS[7:0]}.
If the Main Extension is not implemented, bits [23:8] of the transfer value are RES0. If the Security Extension is not implemented, this value is reserved.

`0b1000000-0b1011111`
FP registers, S0-S31.
If the Floating-point Extension is not implemented, these values are reserved.

All other values are reserved.

If the Floating-point and Security Extensions are implemented, then FPSCR and S0-S31 are not accessible from Non-secure state if DHCSR.S_SDE == 0 and either:

- FPCCR indicates the registers contain values from Secure state.
- NSACR prevents Non-secure access to the registers.

Registers that are not accessible are RAZ/WI.

If this field is written with a reserved value, the PE might behave as if a defined value was written, or ignore the value written, and the value of DCRDR becomes UNKNOWN.

D1.2.34 DDEVARCH, SCS Device Architecture Register

The DDEVARCH characteristics are:

Purpose Provides CoreSight discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

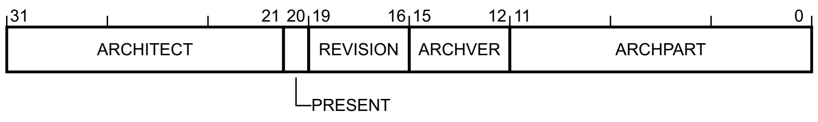
Attributes 32-bit read-only register located at `0xE000EFBC`.

Secure software can access the Non-secure version of this register via DDEVARCH_NS located at `0xE002EFBC`. The location `0xE002EFBC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DDEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

`0x23B` JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

1 DEVARCH information present.

This bit reads as one.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The possible values of this field are:

`0b0000` M-profile debug architecture v3.0.

This field reads as `0b0000`.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

`0b0010`	M-profile debug architecture v3.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0010`.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

`0xA04`	M-profile debug architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA04`.

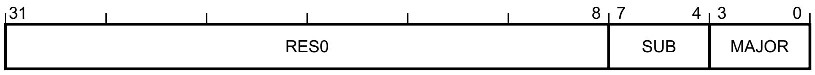
D1.2.35 DDEVTYPE, SCS Device Type Register

The DDEVTYPE characteristics are:

Purpose	Provides CoreSight discovery information for the SCS.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit read-only register located at `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.

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 Halting debug or the Main Extension is implemented.

This register is RES0 if both Halting debug and Main Extension are not implemented.

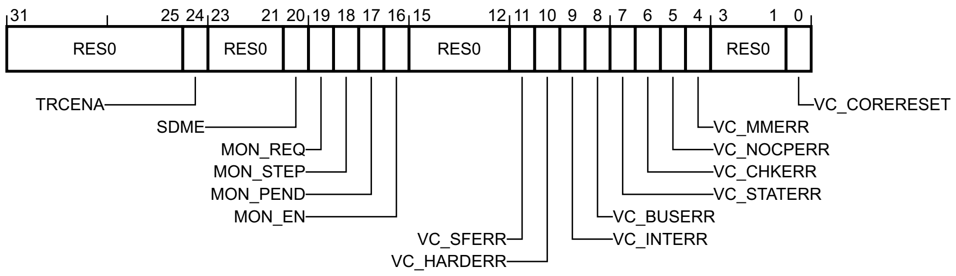
Attributes 32-bit read/write register located at `0xE000EDFC`.

Secure software can access the Non-secure version of this register via DEMCR_NS located at `0xE002EDFC`. The location `0xE002EDFC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DEMCR bit assignments are:



Bits [31:25]

Reserved, RES0.

TRCENA, bit [24]

Trace enable. Global enable for all DWT and ITM features.

The possible values of this bit are:

0	DWT and ITM features disabled.
1	DWT and ITM features enabled.

If the DWT and ITM units are not implemented, this bit is RES0. See the descriptions of DWT and ITM for details of which features this bit controls.

Setting this bit to 0 might not stop all events. To ensure that all events are stopped, software must set all DWT and ITM feature enable bits to 0, and ensure that all trace generated by the DWT and ITM has been flushed, before setting this bit to 0.

It is IMPLEMENTATION DEFINED whether this bit affects how the system processes trace.

Arm recommends that this bit is set to 1 when using an ETM even if any implemented DWT and ITM are not being used.

This bit resets to zero on a Cold reset.

Bits [23:21]

Reserved, RES0.

SDME, bit [20]

Secure DebugMonitor enable. Indicates whether the DebugMonitor targets the Secure or the Non-secure state and whether debug events are allowed in Secure state.

The possible values of this bit are:

0	Debug events prohibited in Secure state and the DebugMonitor exception targets Non-secure state.
1	Debug events allowed in Secure state and the DebugMonitor exception targets Secure state.

When DebugMonitor exception is not pending or active, this bit reflects the value of SecureDebugMonitorAllowed(), otherwise, the previous value is retained.

This bit is read-only.

If the Security Extension is not implemented, this bit is RES0.

If the Main Extension is not implemented, this bit is RES0.

MON_REQ, bit [19]

Monitor request. DebugMonitor semaphore bit.

The PE does not use this bit. The monitor software defines the meaning and use of this bit.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

MON_STEP, bit [18]

Monitor step. Enable DebugMonitor exception stepping.

The possible values of this bit are:

0	Stepping disabled.
1	Stepping enabled.

The effect of changing this bit at an execution priority that is lower than the priority of the DebugMonitor exception is UNPREDICTABLE.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

MON_PEND, bit [17]

Monitor pend. Sets or clears the pending state of the DebugMonitor exception.

The possible values of this bit are:

0	Clear the status of the DebugMonitor exception to not pending.
1	Set the status of the DebugMonitor exception to pending.

When the DebugMonitor exception is pending it becomes active subject to the exception priority rules. The effect of setting this bit to 1 is not affected by the value of the MON_EN bit. This means that software or a debugger can set MON_PEND to 1 and pend a DebugMonitor exception, even when MON_EN is set to 0.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

MON_EN, bit [16]

Monitor enable. Enable the DebugMonitor exception.

The possible values of this bit are:

0	DebugMonitor exception disabled.
1	DebugMonitor exception enabled.

If a debug event halts the PE, the PE ignores the value of this bit.

If DEMCR.SDME is one this bit is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

Bits [15:12]

Reserved, RES0.

VC_SFERR, bit [11]

Vector Catch SecureFault. SecureFault exception Halting debug vector catch enable.

The possible values of this bit are:

0	Halting debug trap on SecureFault disabled.
1	Halting debug trap on SecureFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or DHCSR.S_SDE == 0.

If the Security Extension is not implemented, this bit is RES0.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_HARDERR, bit [10]

Vector Catch HardFault errors. HardFault exception Halting debug vector catch enable.

The possible values of this bit are:

0	Halting debug trap on HardFault disabled.
1	Halting debug trap on HardFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_INTERR, bit [9]

Vector Catch interrupt errors. Enable Halting debug vector catch for faults arising in lazy state preservation and during exception entry or return.

The possible values of this bit are:

0	Halting debug trap on faults disabled.
1	Halting debug trap on faults enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_BUSERR, bit [8]

Vector Catch BusFault errors. BusFault exception Halting debug vector catch enable.

The possible values of this bit are:

0	Halting debug trap on BusFault disabled.
1	Halting debug trap on BusFault enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_STATERR, bit [7]

Vector Catch state errors. Enable Halting debug trap on a UsageFault exception caused by a state information error, for example an Undefined Instruction exception.

The possible values of this bit are:

0	Halting debug trap on UsageFault caused by state information error disabled.
1	Halting debug trap on UsageFault caused by state information error enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_CHKERR, bit [6]

Vector Catch check errors. Enable Halting debug trap on a UsageFault exception caused by a checking error, for example an alignment check error.

The possible values of this bit are:

0	Halting debug trap on UsageFault caused by checking error disabled.
1	Halting debug trap on UsageFault caused by checking error enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_NOCPERR, bit [5]

Vector Catch NOCP errors. Enable Halting debug trap on a UsageFault caused by an access to a coprocessor.

The possible values of this bit are:

0	Halting debug trap on UsageFault caused by access to a coprocessor disabled.
1	Halting debug trap on UsageFault caused by access to a coprocessor enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

VC_MMERR, bit [4]

Vector Catch MemManage errors. Enable Halting debug trap on a MemManage exception.

The possible values of this bit are:

0	Halting debug trap on MemManage disabled.
1	Halting debug trap on MemManage enabled.

The PE ignores the value of this bit if DHCSR.C_DEBUGEN == 0, `HaltingDebugAllowed()` == FALSE, or the Security Extension is implemented, DHCSR.S_SDE == 0 and the exception targets Secure state.

If the Main Extension is not implemented, this bit is RES0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

Bits [3:1]

Reserved, RES0.

VC_CORERESET, bit [0]

Vector Catch Core reset. Enable Reset Vector Catch. This causes a Warm reset to halt a running system.

The possible values of this bit are:

0	Halting debug trap on reset disabled.
1	Halting debug trap on reset enabled.

If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, the PE ignores the value of this bit. Otherwise, when this bit is set to 1 a Warm reset will pend a Vector Catch debug event. The debug event is pended even the PE resets into Secure state and DHCSR.S_SDE == 0.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

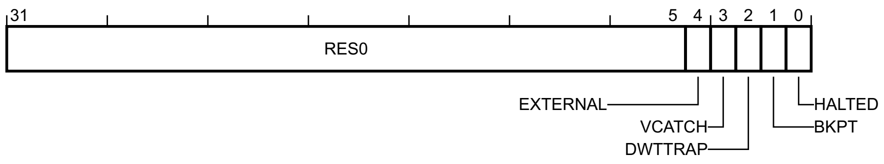
D1.2.37 DFSR, Debug Fault Status Register

The DFSR characteristics are:

Purpose	Shows which debug event occurred.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if Halting debug or the Main Extension is implemented. This register is RES0 if both Halting debug and Main Extension are not implemented.
Attributes	32-bit read/write-one-to-clear register located at `0xE000ED30`. Secure software can access the Non-secure version of this register via DFSR_NS located at `0xE002ED30`. The location `0xE002ED30` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The DFSR bit assignments are:



Bits [31:5]

Reserved, RES0.

EXTERNAL, bit [4]

External event. Sticky flag indicating whether an External debug request debug event has occurred.

The possible values of this bit are:

0	Debug event has not occurred.
1	Debug event has occurred.

This bit resets to zero on a Cold reset.

VCATCH, bit [3]

Vector Catch event. Sticky flag indicating whether a Vector catch debug event has occurred.

The possible values of this bit are:

0	Debug event has not occurred.
1	Debug event has occurred.

If Halting debug is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

DWTTRAP, bit [2]

Watchpoint event. Sticky flag indicating whether a Watchpoint debug event has occurred.

The possible values of this bit are:

0	Debug event has not occurred.

1	Debug event has occurred.

If the DWT is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

BKPT, bit [1]

Breakpoint event. Sticky flag indicating whether a Breakpoint debug event has occurred.

The possible values of this bit are:

0	Debug event has not occurred.
1	Debug event has occurred.

This bit resets to zero on a Cold reset.

HALTED, bit [0]

Halt or step event. Sticky flag indicating that a Halt request debug event or Step debug event has occurred.

The possible values of this bit are:

0	Debug event has not occurred.
1	Debug event has occurred.

This bit resets to zero on a Cold reset.

D1.2.38 DHCSR, Debug Halting Control and Status Register

The DHCSR characteristics are:

Purpose Controls Halting debug.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

It is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software.

Configurations Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

Attributes 32-bit read/write register located at `0xE000EDF0`.

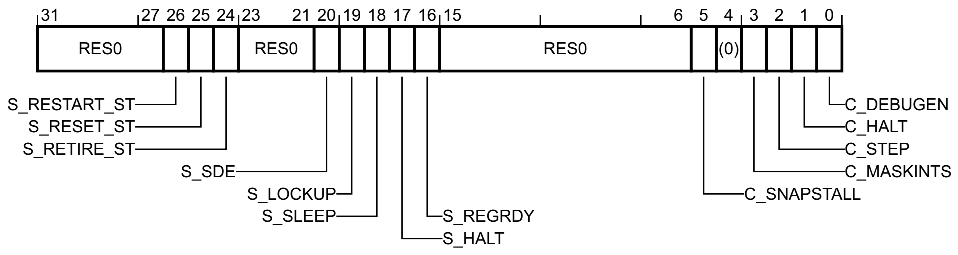
Secure software can access the Non-secure version of this register via DHCSR_NS located at `0xE002EDF0`. The location `0xE002EDF0` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

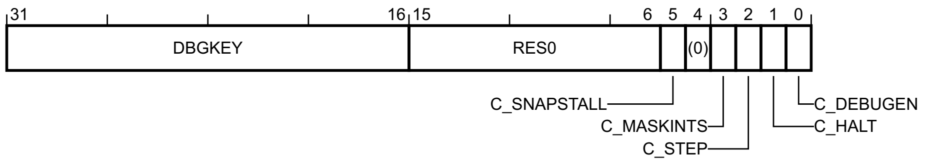
Field descriptions

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.

Bits [23:21], on a read

Reserved, RES0.

S_SDE, bit [20], on a read

Secure debug enabled. Indicates whether Secure invasive debug is allowed.

The possible values of this bit are:

0	Secure invasive debug prohibited.
1	Secure invasive debug allowed.

If the PE is in Non-debug state, this bit reflects the value of 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 [15:6]

Reserved, RES0.

C_SNAPSTALL, bit [5]

Snap stall control. Allow imprecise entry to Debug state.

The possible values of this bit are:

0	No action.
1	Allows imprecise entry to Debug state, for example by forcing any stalled load or store instruction to be abandoned.

Setting this bit to 1 allows a debugger to request an imprecise entry to Debug state. Writing 1 to this bit makes the state of the memory system UNPREDICTABLE. Therefore if a debugger writes 1 to this bit it must reset the system before leaving Debug state.

The effect of setting this bit to 1 is UNPREDICTABLE unless the DHCSR write also sets C_DEBUGEN and C_HALT to 1. This means that if the PE is not already in Debug state, it enters Debug state when the stalled instruction completes.

If the Security Extension is implemented, then writes to this bit are ignored when DHCSR.S_SDE == 0.

If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, the PE ignores this bit and behaves as if it is set to 0.

If the Main Extension is not implemented, this bit is RES0.

Note

A debugger can write to the DHCSR to clear this bit to 0. However, this does not remove the UNPREDICTABLE state of the memory system caused by setting C_SNAPSTALL to 1. The architecture does not guarantee that setting this bit to 1 will force an entry to Debug state. Arm strongly recommends that a value of 1 is never written to C_SNAPSTALL when the PE is in Debug state.

Bit [4]

Reserved, RES0.

C_MASKINTS, bit [3]

Mask interrupts control. When debug is enabled, the debugger can write to this bit to mask PendSV, SysTick and external configurable interrupts.

The possible values of this bit are:

0	Do not mask.
1	Mask PendSV, SysTick and external configurable interrupts.

The effect of any single write to DHCSR that changes the value of this bit is UNPREDICTABLE unless one of:

- Before the write, DHCSR.{S_HALT,C_HALT} are both set to 1 and the write also writes 1 to DHCSR.C_HALT.
- Before the write, DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, and the write writes 0 to DHCSR.C_MASKINTS.

This means that a single write to DHCSR must not clear DHCSR.C_HALT to 0 and change the value of the C_MASKINTS bit.

If the Security Extension is implemented and DHCSR.S_SDE == 0, this bit does not affect interrupts targeting Secure state.

If DHCSR.C_DEBUGEN == 0 or `HaltingDebugAllowed()` == FALSE, the PE ignores this bit and behaves as if it is set to 0.

If DHCSR.C_DEBUGEN == 0 this but 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 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:

- 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.

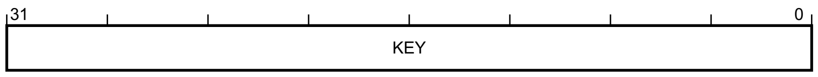
Attributes 32-bit write-only register located at `0xE000EFB0`.

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 this component's registers 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.

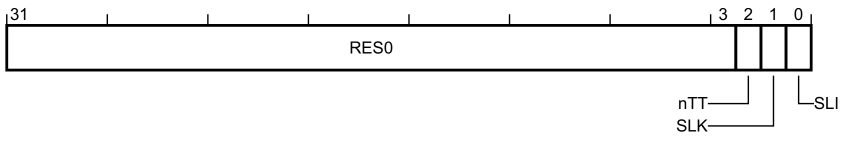
Attributes 32-bit read-only register located at `0xE000EFB4`.

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 discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

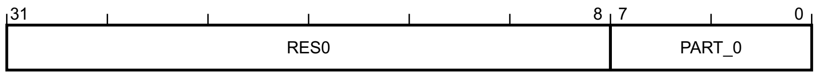
Attributes 32-bit read-only register located at `0xE000EFE0`.

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 discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

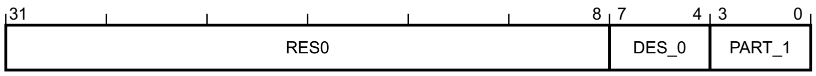
Attributes 32-bit read-only register located at `0xE000EFE4`.

Secure software can access the Non-secure version of this register via DPIDR1_NS located at `0xE002EFE4`. The location `0xE002EFE4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DPIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

PART_1, bits [3:0]

Part number bits [11:8]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.43 DPIDR2, SCS Peripheral Identification Register 2

The DPIDR2 characteristics are:

Purpose Provides CoreSight discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

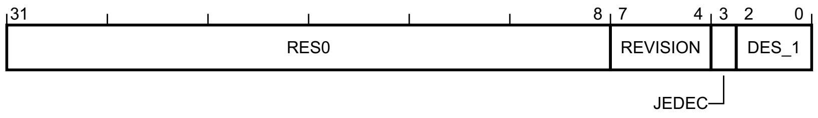
Attributes 32-bit read-only register located at `0xE000EFE8`.

Secure software can access the Non-secure version of this register via DPIDR2_NS located at `0xE002EFE8`. The location `0xE002EFE8` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DPIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component revision. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

JEDEC, bit [3]

JEDEC assignee value is used. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as one.

DES_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

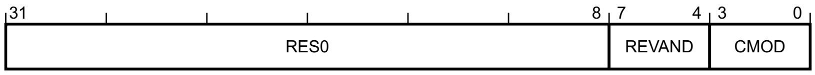
D1.2.44 DPIDR3, SCS Peripheral Identification Register 3

The DPIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the SCS.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit read-only register located at `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.

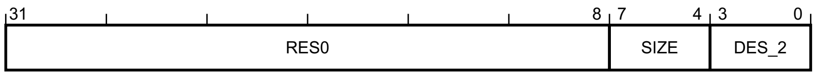
D1.2.45 DPIDR4, SCS Peripheral Identification Register 4

The DPIDR4 characteristics are:

Purpose	Provides CoreSight discovery information for the SCS.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit read-only register located at `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 discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

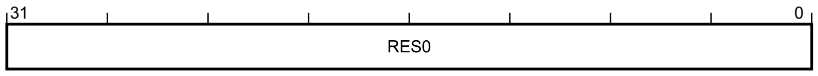
Attributes 32-bit read-only register located at `0xE000EFD4`.

Secure software can access the Non-secure version of this register via DPIDR5_NS located at `0xE002EFD4`. The location `0xE002EFD4` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DPIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

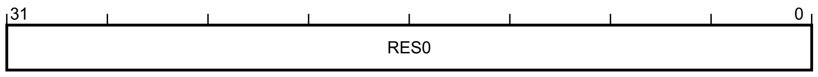
D1.2.47 DPIDR6, SCS Peripheral Identification Register 6

The DPIDR6 characteristics are:

Purpose	Provides CoreSight discovery information for the SCS.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit read-only register located at `0xE000EFD8`. Secure software can access the Non-secure version of this register via DPIDR6_NS located at `0xE002EFD8`. The location `0xE002EFD8` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The DPIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.48 DPIDR7, SCS Peripheral Identification Register 7

The DPIDR7 characteristics are:

Purpose Provides CoreSight discovery information for the SCS.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if Halting debug is implemented.

This register is RES0 if Halting debug is not implemented.

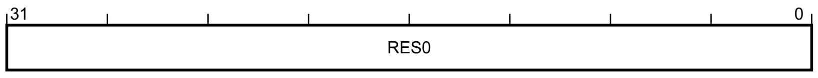
Attributes 32-bit read-only register located at `0xE000EFDC`.

Secure software can access the Non-secure version of this register via DPIDR7_NS located at `0xE002EFDC`. The location `0xE002EFDC` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The DPIDR7 bit assignments are:



Bits [31:0]

Reserved, RES0.

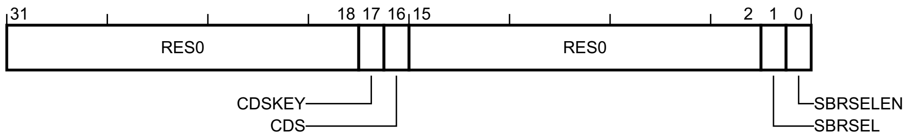
D1.2.49 DSCSR, Debug Security Control and Status Register

The DSCSR characteristics are:

Purpose	Provides control and status information for Secure debug.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. 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 the Security Extension is implemented. This register is RES0 if the Security Extension is not implemented. Present only if Halting debug is implemented. This register is RES0 if Halting debug is not implemented.
Attributes	32-bit read/write register located at `0xE000EE08`. This register is not banked between Security states.

Field descriptions

The DSCSR bit assignments are:



Bits [31:18]

Reserved, RES0.

CDSKEY, bit [17]

CDS write-enable key. Writes to the CDS bit are ignored unless CDSKEY is concurrently written to zero.

The possible values of this bit are:

0	Concurrent write to CDS not ignored.
1	Concurrent write to CDS ignored.

This bit reads-as-one.

CDS, bit [16]

Current domain Secure. This field indicates the current Security state of the processor.

The possible values of this bit are:

0	PE is in Non-secure state.
1	PE is in Secure state.

This bit is only writable if DHCSR.S_SDE is 1, the access to the register originates from the debugger, the PE is halted in Debug state, and CDSKEY is concurrently written to zero.

Bits [15:2]

Reserved, RES0.

SBRSEL, bit [1]

Secure banked register select. If SBRSELEN is 1 this bit selects whether the Non-secure or the Secure versions of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

0	Selects the Non-secure versions.
1	Selects the Secure versions.

This bit behaves as RAZ/WI if DHCSR.S_SDE is 0.

This bit resets to zero on a Cold reset.

SBRSELEN, bit [0]

Secure banked register select enable. Controls whether the SBRSEL field or the current Security state of the processor selects which version of the memory-mapped banked registers are accessible to the debugger.

The possible values of this bit are:

0	The current Security state of the PE determines which memory-mapped Banked registers are accessed by the debugger.
1	DSCSR.SBRSEL selects which memory-mapped Banked registers are accessed by the debugger.

This bit behaves as RAO/WI if DHCSR.S_SDE is 0.

This bit resets to zero on a Cold reset.

Note

This method of banked register selection means that the register aliasing is not used for accesses from the debugger. Accesses to the aliased addresses from the debugger have the same behavior as reserved addresses.

D1.2.50 DWT_CIDR0, DWT Component Identification Register 0

The DWT_CIDR0 characteristics are:

Purpose Provides CoreSight discovery information for the DWT.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

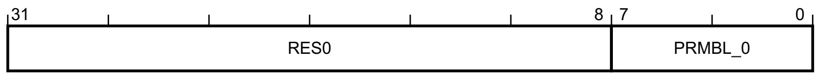
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read-only register located at `0xE0001FF0`.

This register is not banked between Security states.

Field descriptions

The DWT_CIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

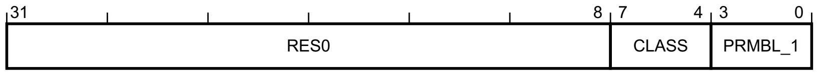
D1.2.51 DWT_CIDR1, DWT Component Identification Register 1

The DWT_CIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FF4`. This register is not banked between Security states.

Field descriptions

The DWT_CIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

CoreSight component class. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

PRMBL_1, bits [3:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

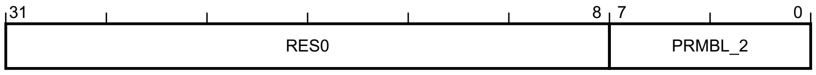
D1.2.52 DWT_CIDR2, DWT Component Identification Register 2

The DWT_CIDR2 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FF8`. This register is not banked between Security states.

Field descriptions

The DWT_CIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

D1.2.53 DWT_CIDR3, DWT Component Identification Register 3

The DWT_CIDR3 characteristics are:

Purpose Provides CoreSight discovery information for the DWT.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

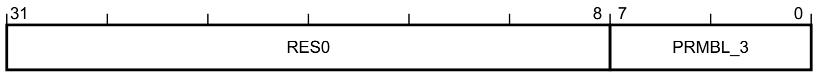
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read-only register located at `0xE0001FFC`.

This register is not banked between Security states.

Field descriptions

The DWT_CIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

D1.2.54 DWT_COMPn, DWT Comparator Register, n = 0 - 14

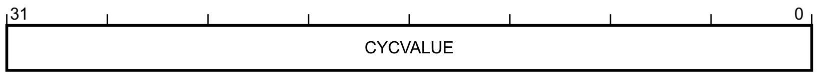
The DWT_COMP{0..14} characteristics are:

Purpose	Provides a reference value for use by watchpoint comparator *n*.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read/write register located at `0xE0001020` + 16*n*. This register is not banked between Security states.

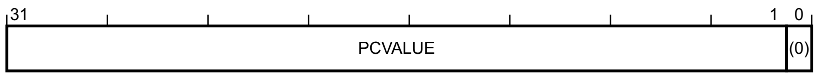
Field descriptions

The DWT_COMP{0..14} bit assignments are:

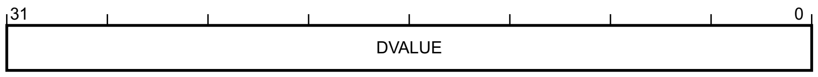
When `DWT_FUNCTIONn.MATCH == 0b0001`:



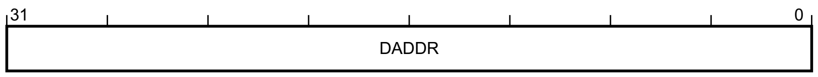
When `DWT_FUNCTIONn.MATCH == 0b001x`:



When `DWT_FUNCTIONn.MATCH == 0b10xx`:



When `DWT_FUNCTIONn.MATCH == 0bx1xx`:



CYCVALUE, bits [31:0], when DWT_FUNCTIONn.MATCH == `0b0001`

Cycle value. Reference value for comparison with cycle count.

This field resets to an UNKNOWN value on a Cold reset.

PCVALUE, bits [31:1], when DWT_FUNCTIONn.MATCH == `0b001x`

PC value. Reference value for comparison with Program Counter.

This field resets to an UNKNOWN value on a Cold reset.

Bit [0], when DWT_FUNCTIONn.MATCH == `0b001x`

Reserved, RES0.

DADDR, bits [31:0], when DWT_FUNCTIONn.MATCH == `0bx1xx`

Data address. Reference value for comparison with load or store address.

For halfword address comparisons, DADDR[0] is RES0. For byte address comparisons, DADDR[1:0] are RES0.

This field resets to an UNKNOWN value on a Cold reset.

DVALUE, bits [31:0], when DWT_FUNCTIONn.MATCH == `0b10xx`

Data value. Reference value for comparison with load or store data.

For halfword or word comparisons, the data value is in little-endian order. That is, the least significant byte of this register is compared with the byte targeting the lowest address in memory.

For byte or halfword comparisons, if the value of the byte or halfword is not replicated across all byte or halfword lanes, the value used for the comparison is UNKNOWN.

This field resets to an UNKNOWN value on a Cold reset.

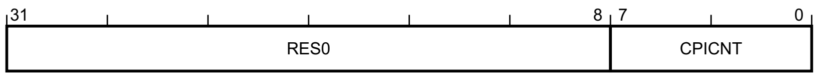
D1.2.55 DWT_CPICNT, DWT CPI Count Register

The DWT_CPICNT characteristics are:

Purpose	Counts additional cycles required to execute multicycle instructions and instruction fetch stalls.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented. Present only if DWT_CTRL.NOPRFCNT == 0. This register is RES0 if DWT_CTRL.NOPRFCNT == 1.
Attributes	32-bit read/write register located at `0xE0001008`. This register is not banked between Security states.

Field descriptions

The DWT_CPICNT bit assignments are:



Bits [31:8]

Reserved, RES0.

CPICNT, bits [7:0]

Base instruction overhead counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.CPIEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed.
- No load-store operation is in progress, see DWT_LSUCNT.
- No exception-entry or exception-exit operation is in progress, see DWT_EXCCNT.
- The PE is not in a power-saving mode, see DWT_SLEEPCNT.
- Either `SecureNoninvasiveDebugAllowed()` == TRUE, or the PE is in Non-secure state and `NoninvasiveDebugAllowed()` == TRUE.

The definition of "no instruction is executed" is IMPLEMENTATION DEFINED. Arm recommends that this counts each cycle on which no instruction is retired.

Initialized to zero when the counter is disabled and DWT_CTRL.CPIEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

D1.2.56 DWT_CTRL, DWT Control Register

The DWT_CTRL characteristics are:

Purpose Provides configuration and status information for the DWT unit, and used to control features of the unit.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if the DWT is implemented.

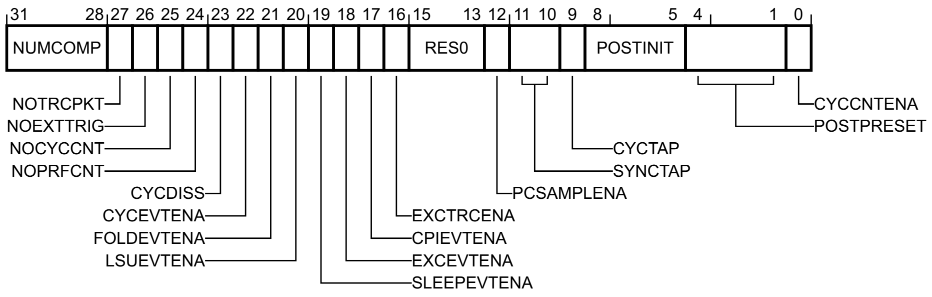
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read/write register located at `0xE0001000`.

This register is not banked between Security states.

Field descriptions

The DWT_CTRL bit assignments are:



NUMCOMP, bits [31:28]

Number of comparators. Number of DWT comparators implemented.

A value of zero indicates no comparator support.

This field reads as an IMPLEMENTATION DEFINED value.

NOTRCPKT, bit [27]

No trace packets. Indicates whether the implementation does not support trace.

The possible values of this bit are:

0	Trace supported.
1	Trace not supported.

If this bit is RAZ, the NOCYCCNT bit must also RAZ.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

NOEXTTRIG, bit [26]

No External Triggers. Shows whether the implementation does not support external triggers.

Reserved, RES0.

NOCYCCNT, bit [25]

No cycle count. Indicates whether the implementation does not include a cycle counter.

The possible values of this bit are:

0	Cycle counter implemented.
1	Cycle counter not implemented.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

NOPRFCNT, bit [24]

No profile counters. Indicates whether the implementation does not include the profiling counters.

The possible values of this bit are:

0	Profiling counters implemented.
1	Profiling counters not implemented.

If the Main Extension is not implemented, this bit is RES1.

This bit reads as an IMPLEMENTATION DEFINED value.

CYCDISS, bit [23]

Cycle counter disabled secure. Controls whether the cycle counter is disabled in Secure state.

The possible values of this bit are:

0	No effect.
1	Disable incrementing of the cycle counter when the PE is in Secure state.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

CYCEVTENA, bit [22]

Cycle event enable. Enables Event Counter packet generation on POSTCNT underflow.

The possible values of this bit are:

0	No Event Counter packets generated when POSTCNT underflows.
1	If PCSAMPLENA set to 0, an Event Counter packet is generated when POSTCNT underflows.

RES0 if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

FOLDEVTENA, bit [21]

Fold event enable. Enables DWT_FOLDCNT counter.

The possible values of this bit are:

0	DWT_FOLDCNT disabled.
1	DWT_FOLDCNT enabled.

RES0 if the NOPRFCNT bit is RAO. The reset value is 0.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

LSUEVTENA, bit [20]

LSU event enable. Enables DWT_LSUCNT counter.

The possible values of this bit are:

0	DWT_LSUCNT disabled.
1	DWT_LSUCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

SLEEPEVTENA, bit [19]

Sleep event enable. Enable DWT_SLEEPCNT counter.

The possible values of this bit are:

0	DWT_SLEEPCNT disabled.
1	DWT_SLEEPCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

EXCEVTENA, bit [18]

Exception event enable. Enables DWT_EXCCNT counter.

The possible values of this bit are:

0	DWT_EXCCNT disabled.
1	DWT_EXCCNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

CPIEVTENA, bit [17]

CPI event enable. Enables DWT_CPICNT counter.

The possible values of this bit are:

0	DWT_CPICNT disabled.
1	DWT_CPICNT enabled.

RES0 if the NOPRFCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

EXCTRCENA, bit [16]

Exception trace enable. Enables generation of Exception Trace packets.

The possible values of this bit are:

0	Exception Trace packet generation disabled.
1	Exception Trace packet generation enabled.

RES0 if the NOTRCPKT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

Bits [15:13]

Reserved, RES0.

PCSAMPLENA, bit [12]

PC sample enable. Enables use of POSTCNT counter as a timer for Periodic PC Sample packet generation.

The possible values of this bit are:

0	Periodic PC Sample packet generation disabled.
1	Periodic PC Sample packet generated on POSTCNT underflow.

RES0 if the NOTRCPKT bit is RAO or the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

SYNCTAP, bits [11:10]

Synchronization tap. Selects the position of the synchronization packet request counter tap on the CYCCNT counter. This determines the rate of Synchronization packet requests made by the DWT.

The possible values of this field are:

`0b00`	Synchronization packet request disabled.
`0b01`	Synchronization counter tap at CYCCNT[24].
`0b10`	Synchronization counter tap at CYCCNT[26].
`0b11`	Synchronization counter tap at CYCCNT[28].

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

CYCTAP, bit [9]

Cycle count tap. Selects the position of the POSTCNT tap on the CYCCNT counter.

The possible values of this bit are:

0	POSTCNT tap at CYCCNT[6].
1	POSTCNT tap at CYCCNT[10].

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to an UNKNOWN value on a Cold reset.

POSTINIT, bits [8:5]

POSTCNT initial. Initial value for the POSTCNT counter.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

POSTPRESET, bits [4:1]

POSTCNT preset. Reload value for the POSTCNT counter.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this field is RES0.

This field resets to an UNKNOWN value on a Cold reset.

CYCCNTENA, bit [0]

CYCCNT enable. Enables CYCCNT.

The possible values of this bit are:

0	CYCCNT disabled.
1	CYCCNT enabled.

RES0 if the NOCYCCNT bit is RAO.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Cold reset.

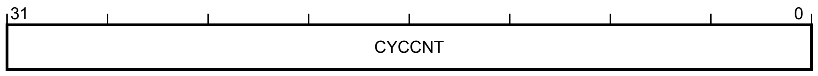
D1.2.57 DWT_CYCCNT, DWT Cycle Count Register

The DWT_CYCCNT characteristics are:

Purpose	Shows or sets the value of the processor cycle counter, CYCCNT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented. Present only if DWT_CTRL.NOCYCCNT == 0. This register is RES0 if DWT_CTRL.NOCYCCNT == 1.
Attributes	32-bit read/write register located at `0xE0001004`. This register is not banked between Security states.

Field descriptions

The DWT_CYCCNT bit assignments are:



CYCCNT, bits [31:0]

Incrementing cycle counter value. Increments one on each processor clock cycle when DWT_CTRL.CYCCNTENA == 1 and DEMCR.TRCENA == 1. On overflow, CYCCNT wraps to zero.

This field resets to an UNKNOWN value on a Cold reset.

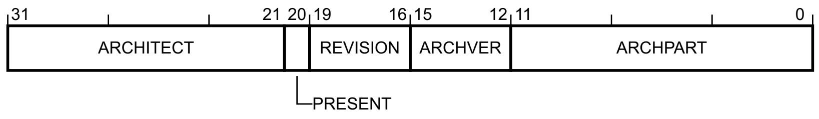
D1.2.58 DWT_DEVARCH, DWT Device Architecture Register

The DWT_DEVARCH characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FBC`. This register is not banked between Security states.

Field descriptions

The DWT_DEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

`0x23B`	JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

1	DEVARCH information present.

This bit reads as one.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The possible values of this field are:

`0b0000`	DWT architecture v2.0.

This field reads as `0b0000`.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

`0b0001`	DWT architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0001`.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

`0xA02`	DWT architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA02`.

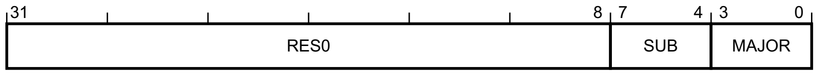
D1.2.59 DWT_DEVTYPE, DWT Device Type Register

The DWT_DEVTYPE characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `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`.

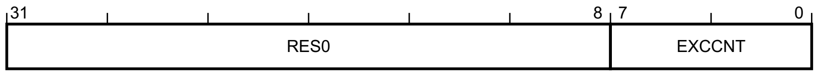
D1.2.60 DWT_EXCCNT, DWT Exception Overhead Count Register

The DWT_EXCCNT characteristics are:

Purpose	Counts the total cycles spent in exception processing.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented. Present only if DWT_CTRL.NOPRFCNT == 0. This register is RES0 if DWT_CTRL.NOPRFCNT == 1.
Attributes	32-bit read/write register located at `0xE000100C`. This register is not banked between Security states.

Field descriptions

The DWT_EXCCNT bit assignments are:



Bits [31:8]

Reserved, RES0.

EXCCNT, bits [7:0]

The exception overhead counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.EXCEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed, see DWT_CPICNT.
- An exception-entry or exception-exit related operation is in progress.
- Either `SecureNoninvasiveDebugAllowed()` == TRUE, or NS-Req for the operation is set to Non-secure and `NoninvasiveDebugAllowed()` == TRUE.

Exception-entry or exception-exit related operations include the stacking of registers on exception entry, lazy state preservation, unstacking of registers on exception exit, and preemption.

Initialized to zero when the counter is disabled and DWT_CTRL.EXCEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

D1.2.61 DWT_FOLDCNT, DWT Folded Instruction Count Register

The DWT_FOLDCNT characteristics are:

Purpose Increments for each additional instruction executed in the current cycle.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Present only if the DWT is implemented.

This register is RES0 if the DWT is not implemented.

Present only if DWT_CTRL.NOPRFCNT == 0.

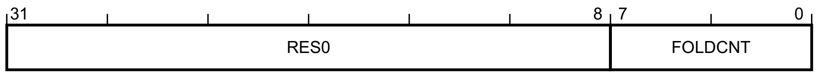
This register is RES0 if DWT_CTRL.NOPRFCNT == 1.

Attributes 32-bit read/write register located at `0xE0001018`.

This register is not banked between Security states.

Field descriptions

The DWT_FOLDCNT bit assignments are:



Bits [31:8]

Reserved, RES0.

FOLDCNT, bits [7:0]

Folded instruction counter.

Counts on each cycle when all of the following are true:

- DWT_CTRL.FOLDEVTENA == 1 and DEMCR.TRCENA == 1.
- At least two instructions are executed, see DWT_CPICNT.
- Either `SecureNoninvasiveDebugAllowed()` == TRUE, or the PE is in Non-secure state and `NoninvasiveDebugAllowed()` == TRUE.

The counter is incremented by the number of instructions executed, minus one.

Initialized to zero when the counter is disabled and DWT_CTRL.FOLDEVTENA is written with 1. An event is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

D1.2.62 DWT_FUNCTIONn, DWT Comparator Function Register, n = 0 - 14

The DWT_FUNCTION{0..14} characteristics are:

Purpose Controls the operation of watchpoint comparator *n*.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if the DWT is implemented.

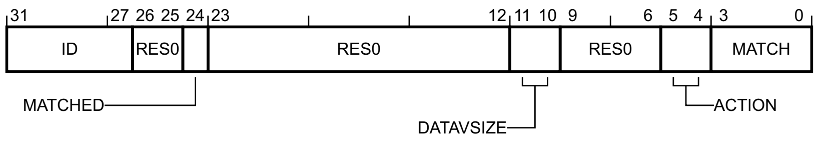
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read/write register located at `0xE0001028` + 16*n*.

This register is not banked between Security states.

Field descriptions

The DWT_FUNCTION{0..14} bit assignments are:



ID, bits [31:27]

Identify capability. Identifies the capabilities for MATCH for comparator *n*.

The possible values of this field are:

`0b00000`	Reserved.
`0b01000`	Data Address, and Data Address With Value.
`0b01001`	Cycle Counter, Data Address, and Data Address With Value.
`0b01010`	Instruction Address, Data Address, and Data Address With Value.
`0b01011`	Cycle Counter, Instruction Address, Data Address and Data Address With Value.
`0b11000`	Data Address, Data Address Limit, and Data Address With Value.
`0b11010`	Instruction Address, Instruction Address Limit, Data Address, Data Address Limit, and Data Address With Value.
`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 are only supported if the Main Extension is 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* [15:18] must be equal to DWT_COMP*n* [7:0].

This field resets to an UNKNOWN value on a Cold reset.

Bits [9:6]

Reserved, RES0.

ACTION, bits [5:4]

Action on match. Defines the action on a match. This field is ignored and the comparator generates no actions if it is disabled by MATCH.

The possible values of this field are:

`0b00`	Trigger only.
`0b01`	Generate debug event.
`0b10`	For a Cycle Counter, Instruction Address, Data Address, Data Value or Linked Data Value comparator, generate a Data Trace Match packet. For a Data Address With Value comparator, generate a Data Trace Data Value packet.
`0b11`	For a Data Address Limit comparator, generate a Data Trace Data Address packet. For a Cycle Counter, Instruction Address Limit, or Data Address comparator, generate a Data Trace PC Value packet.

For a Data Address With Value comparator, generate both a Data Trace PC Value packet and a Data Trace Data Value packet.

If the Main Extension is not implemented, the values `0b10` and `0b11` are reserved.

This field resets to an UNKNOWN value on a Cold reset.

MATCH, bits [3:0]

Match type. Controls the type of match generated by this comparator.

The possible values of this field are:

`0b0000`	Disabled. Never generates a match.
`0b0001`	Cycle Counter. Matches if DWT_CYCCNT equals the comparator value. The comparator is checked each time DWT_CYCCNT is written to, directly or indirectly. Only supported if the Main Extension is implemented, DWT_FUNCTION<n>.ID<0> == 1 and DWT_CTRL.NOCYCCNT == 0.
`0b0010`	Instruction Address. If not linked to, an instruction matches if the address of the first byte of the instruction matches the comparator address. Only supported if DWT_FUNCTION<n>.ID<1> == 1.
`0b0011`	Instruction Address Limit. An instruction matches if the address of the first byte of the instruction lies between the lower comparator address (specified by comparator <*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.

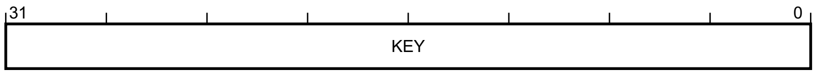
D1.2.63 DWT_LAR, DWT Software Lock Access Register

The DWT_LAR characteristics are:

Purpose	Provides CoreSight Software Lock control for the DWT, see the *ARM® CoreSight™ Architecture Specification* for details.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
	This register is RAZ/WI if accessed via the debugger.
Configurations	Present only if CoreSight identification is implemented.
	This register is RES0 if CoreSight identification is not implemented.
	Present only if the DWT is implemented.
	This register is RES0 if the DWT is not implemented.
	Present only if the optional Software Lock is implemented.
	This register is RAZ/WI if the Software Lock is not implemented.
Attributes	32-bit write-only register located at `0xE0001FB0`.
	This register is not banked between Security states.

Field descriptions

The DWT_LAR bit assignments are:



KEY, bits [31:0]

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

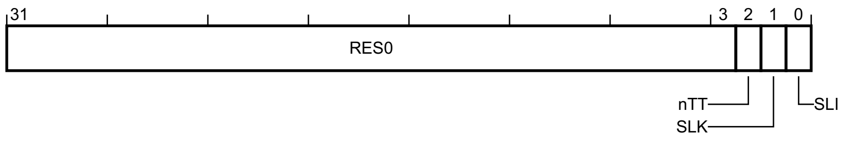
D1.2.64 DWT_LSR, DWT Software Lock Status Register

The DWT_LSR characteristics are:

Purpose	Provides CoreSight Software Lock status information for the DWT, see the *ARM® CoreSight™ Architecture Specification* for details.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. This register is RAZ/WI if accessed via the debugger.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented. Present only if the optional Software Lock is implemented. This register is RAZ/WI if the Software Lock is not implemented.
Attributes	32-bit read-only register located at `0xE0001FB4`. This register is not banked between Security states.

Field descriptions

The DWT_LSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as zero.

SLK, bit [1]

Software Lock status. See the *ARM® CoreSight™ Architecture Specification*.

The possible values of this bit are:

0	Lock clear. Software writes are permitted to this component's registers.
1	Lock set. Software writes to this component's registers 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.

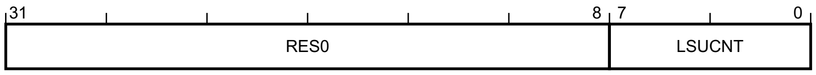
D1.2.65 DWT_LSUCNT, DWT LSU Count Register

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.
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.

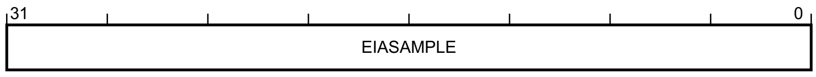
D1.2.66 DWT_PCSR, DWT Program Counter Sample Register

The DWT_PCSR characteristics are:

Purpose	Samples the current value of the Program Counter.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE000101C`. This register is not banked between Security states.

Field descriptions

The DWT_PCSR bit assignments are:



EIASAMPLE, bits [31:0]

Executed instruction address sample. Recently executed instruction address sample value.

The possible values of this field are:

`0xFFFFFFFF`

One of the following is true:

- The PE is halted in Debug state.
- The Security Extension is implemented, the sampled instruction was executed in Secure state, and `SecureNoninvasiveDebugAllowed()` == FALSE.
- `NoninvasiveDebugAllowed()` == FALSE.
- DEMCR.TRCENA == 0.
- The address of a recently-executed instruction is not available.

Not `0xFFFFFFFF`

Instruction address of a recently executed instruction. Bit [0] of the sample instruction address is 0.

The conditions when the address of a recently-executed instruction is not available are IMPLEMENTATION DEFINED.

D1.2.67 DWT_PIDR0, DWT Peripheral Identification Register 0

The DWT_PIDR0 characteristics are:

Purpose Provides CoreSight discovery information for the DWT.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

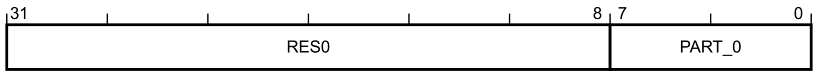
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read-only register located at `0xE0001FE0`.

This register is not banked between Security states.

Field descriptions

The DWT_PIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number bits [7:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

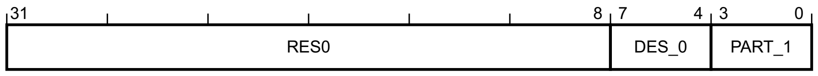
D1.2.68 DWT_PIDR1, DWT Peripheral Identification Register 1

The DWT_PIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
	If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented.
	This register is RES0 if CoreSight identification is not implemented.
	Present only if the DWT is implemented.
	This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FE4`.
	This register is not banked between Security states.

Field descriptions

The DWT_PIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

PART_1, bits [3:0]

Part number bits [11:8]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

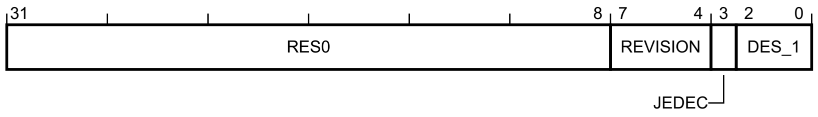
D1.2.69 DWT_PIDR2, DWT Peripheral Identification Register 2

The DWT_PIDR2 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FE8`. This register is not banked between Security states.

Field descriptions

The DWT_PIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component revision. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

JEDEC, bit [3]

JEDEC assignee value is used. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as one.

DES_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

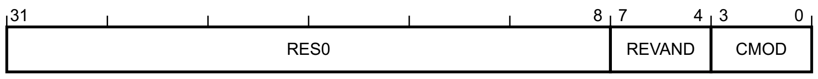
D1.2.70 DWT_PIDR3, DWT Peripheral Identification Register 3

The DWT_PIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FEC`. This register is not banked between Security states.

Field descriptions

The DWT_PIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

RevAnd. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

CMOD, bits [3:0]

Customer Modified. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

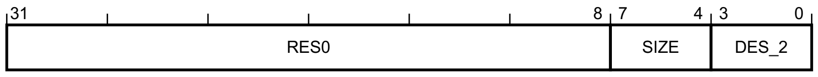
D1.2.71 DWT_PIDR4, DWT Peripheral Identification Register 4

The DWT_PIDR4 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FD0`. This register is not banked between Security states.

Field descriptions

The DWT_PIDR4 bit assignments are:



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

4KB count. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as zero.

DES_2, bits [3:0]

JEP106 continuation code. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

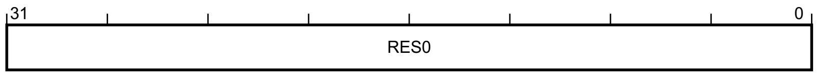
D1.2.72 DWT_PIDR5, DWT Peripheral Identification Register 5

The DWT_PIDR5 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FD4`. This register is not banked between Security states.

Field descriptions

The DWT_PIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.73 DWT_PIDR6, DWT Peripheral Identification Register 6

The DWT_PIDR6 characteristics are:

Purpose Provides CoreSight discovery information for the DWT.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the DWT is implemented.

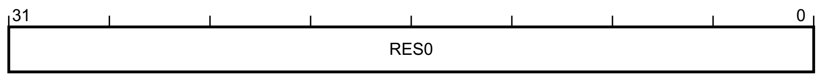
This register is RES0 if the DWT is not implemented.

Attributes 32-bit read-only register located at `0xE0001FD8`.

This register is not banked between Security states.

Field descriptions

The DWT_PIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

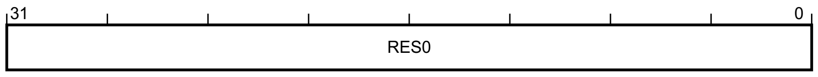
D1.2.74 DWT_PIDR7, DWT Peripheral Identification Register 7

The DWT_PIDR7 characteristics are:

Purpose	Provides CoreSight discovery information for the DWT.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented.
Attributes	32-bit read-only register located at `0xE0001FDC`. This register is not banked between Security states.

Field descriptions

The DWT_PIDR7 bit assignments are:



Bits [31:0]

Reserved, RES0.

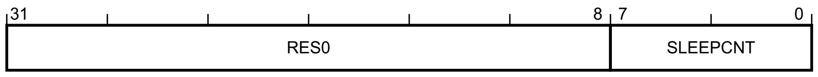
D1.2.75 DWT_SLEEPCNT, DWT Sleep Count Register

The DWT_SLEEPCNT characteristics are:

Purpose	Counts the total number of cycles that the processor is sleeping.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented. Present only if the DWT is implemented. This register is RES0 if the DWT is not implemented. Present only if DWT_CTRL.NOPRFCNT == 0. This register is RES0 if DWT_CTRL.NOPRFCNT == 1.
Attributes	32-bit read/write register located at `0xE0001010`. This register is not banked between Security states.

Field descriptions

The DWT_SLEEPCNT bit assignments are:



Bits [31:8]

Reserved, RES0.

SLEEPCNT, bits [7:0]

Sleep counter.

Counts one on each cycle when all of the following are true:

- DWT_CTRL.SLEEPEVTENA == 1 and DEMCR.TRCENA == 1.
- No instruction is executed, see DWT_CPICNT.
- No load-store operation is in progress, see DWT_LSUCNT.
- No exception-entry or exception-exit operation is in progress, see DWT_EXCCNT.
- The PE is in a power-saving mode.
- Either `SecureNoninvasiveDebugAllowed()` == TRUE, or the PE is in Non-secure state and `NoninvasiveDebugAllowed()` == TRUE.

Power-saving modes include WFI, WFE, and Sleep-on-exit.

All power-saving features are IMPLEMENTATION DEFINED and therefore when this counter counts is IMPLEMENTATION DEFINED. In particular, it is IMPLEMENTATION DEFINED whether the counter increments if the PE is in a power-saving mode and SCR.SLEEPDEEP is set.

Initialized to zero when the counter is disabled and DWT_CTRL.SLEEPEVTENA is written with 1. An Event Counter packet is emitted on counter overflow.

This field resets to an UNKNOWN value on a Cold reset.

Note

Arm recommends that this counter counts all cycles when the PE is sleeping and SCR.SLEEPDEEP is clear, regardless of whether a WFI or WFE instruction, or Sleep-on-exit, caused the entry to the power-saving mode.

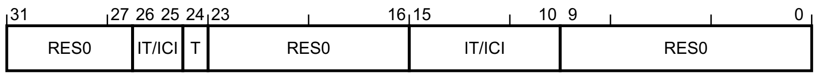
D1.2.76 EPSR, Execution Program Status Register

The EPSR characteristics are:

Purpose	Holds Execution state bits.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The EPSR bit assignments are:



Bits [31:27]

Reserved, RES0.

T, bit [24]

T32 state bit. Determines the current instruction set state.

The possible values of this bit are:

0	Execution of any instruction generates an INVSTATE UsageFault.
1	Instructions decoded as T32 instructions.

This bit resets to an UNKNOWN value on a Warm reset.

Bits [23:16]

Reserved, RES0.

IT/ICI, bits [15:10, 26:25]

If-then and interrupt continuation. Depending on value, this field encodes either the current condition and position in an IT block sequence, or information on the outstanding register list for an interrupted exception-continuable multicycle load or store instruction.

The field IT[7:0] is equivalent to EPSR[15:10,26:25]. The field ICI[7:0] is equivalent to EPSR[26:25,15:10].

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

Bits [9:0]

Reserved, RES0.

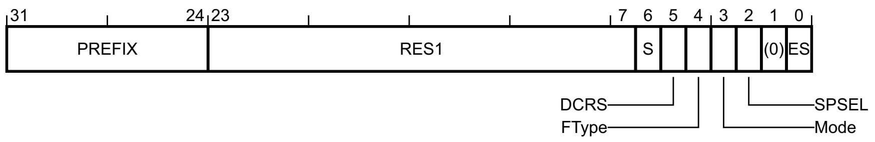
D1.2.77 EXC_RETURN, Exception Return Payload

The EXC_RETURN characteristics are:

Purpose	Value provided in LR on entry to an exception, and used with a BX or load to PC to perform an exception return.
Usage constraints	None.
Configurations	All.
Attributes	32-bit payload.

Field descriptions

The EXC_RETURN bit assignments are:



PREFIX, bits [31:24]

Prefix. Indicates that this is an EXC_RETURN value.

This field reads as `0b11111111`.

Bits [23:7]

Reserved, RES1.

S, bit [6]

Secure or Non-secure stack. Indicates whether a Secure or Non-secure stack is used to restore stack frame on exception return.

The possible values of this bit are:

0	Non-secure stack used.
1	Secure stack used.

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.

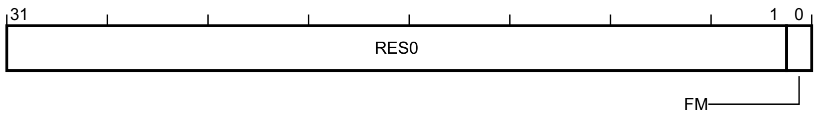
D1.2.78 FAULTMASK, Fault Mask Register

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.

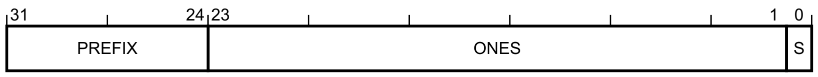
D1.2.79 FNC_RETURN, Function Return Payload

The FNC_RETURN characteristics are:

Purpose	Value provided in LR on entry to Non-secure state from a Secure BLXNS.
Usage constraints	None.
Configurations	All.
Attributes	32-bit payload.

Field descriptions

The FNC_RETURN bit assignments are:



PREFIX, bits [31:24]

This field reads as `0b11111110`.

ONES, bits [23:1]

This field reads as `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 Secure state.
1	From Non-secure state.

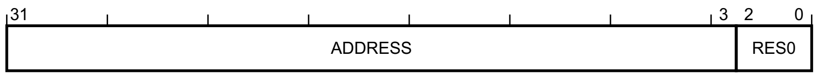
D1.2.80 FPCAR, Floating-Point Context Address Register

The FPCAR characteristics are:

Purpose	Holds the location of the unpopulated floating-point register space allocated on an exception stack frame.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Floating-point Extension is implemented. This register is RES0 if the Floating-point Extension is not implemented.
Attributes	32-bit read/write register located at `0xE000EF38`. Secure software can access the Non-secure version of this register via FPCAR_NS located at `0xE002EF38`. The location `0xE002EF38` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Field descriptions

The FPCAR bit assignments are:



ADDRESS, bits [31:3]

Address. The location of the unpopulated floating-point register space allocated on an exception stack frame.

This field resets to an UNKNOWN value on a Warm reset.

Bits [2:0]

Reserved, RES0.

D1.2.81 FPCCR, Floating-Point Context Control Register

The FPCCR characteristics are:

Purpose Holds control data for the Floating Point Unit.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

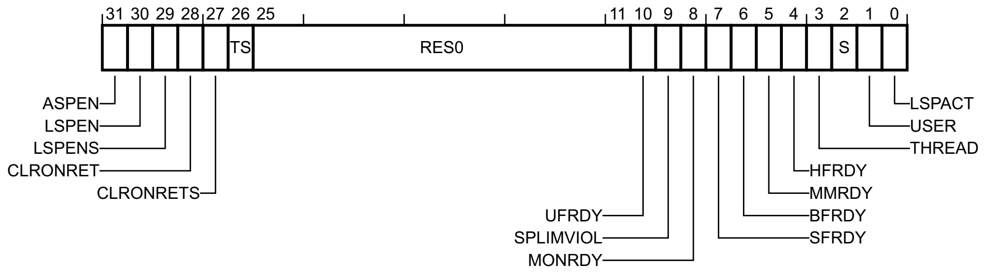
Attributes 32-bit read/write register located at `0xE000EF34`.

Secure software can access the Non-secure version of this register via FPCCR_NS located at `0xE002EF34`. The location `0xE002EF34` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The FPCCR bit assignments are:



ASPEN, bit [31]

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 FP context on exception entry and restores it on exception return.

This bit resets to one on a Warm reset.

LSPEN, bit [30]

Enables lazy context save of FP state.

The possible values of this bit are:

0	Disable automatic lazy context save.
1	Enable automatic lazy context save.

This bit resets to one on a Warm reset.

LSPENS, bit [29]

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 floating point caller saved registers on exception return.

The possible values of this bit are:

0	Disabled.
1	Enabled.

When set to 1 the caller saved floating-point registers (S0 to S15, and FPSCR) are cleared on exception return (including tail chaining) if CONTROL.FPCA is set to 1 and FPCCR_S.LSPACT is set to 0.

This bit resets to zero on a Warm reset.

CLRONRETS, bit [27]

CLRONRET Secure only.

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 FP 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 processor is in the 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. Because 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]

Indicates whether the software executing when the processor 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]

This bit is banked between the Security states and 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]

Indicates whether the software executing when the processor 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]

Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the SecureFault exception to pending. This bit is only present in the Secure version of the register, and behaves as RAZ/WI when accessed from the Non-secure state.

This bit is RAZ/WI from Non-secure state.

This bit resets to an UNKNOWN value on a Warm reset.

BFRDY, bit [6]

Indicates whether the software executing when the processor 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.

This bit resets to an UNKNOWN value on a Warm reset.

MMRDY, bit [5]

Indicates whether the software executing when the processor 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]

Indicates whether the software executing when the processor 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.

This bit resets to an UNKNOWN value on a Warm reset.

THREAD, bit [3]

Indicates the processor 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 status of the floating-point context. This bit is only present in the Secure version of the register, and behaves as RAZ/WI when accessed from the Non-secure state. 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]

Indicates the privilege level of the software executing when the processor 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]

Indicates whether lazy preservation of the floating-point state is active.

This bit is banked between Security states.

The possible values of this bit are:

0	Lazy state preservation is not active.
1	Lazy state preservation is active.

This bit resets to zero on a Warm reset.

D1.2.82 FPDSCR, Floating-Point Default Status Control Register

The FPDSCR characteristics are:

Purpose Holds the default values for the floating-point status control data that the PE assigns to FPSCR when it creates a new floating-point context.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Floating-point Extension is implemented.

This register is RES0 if the Floating-point Extension is not implemented.

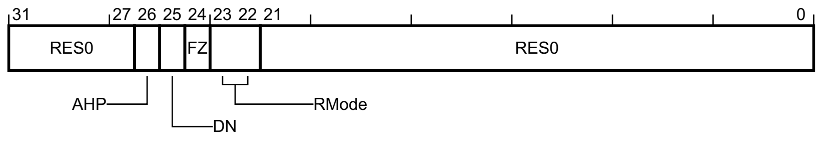
Attributes 32-bit read/write register located at `0xE000EF3C`.

Secure software can access the Non-secure version of this register via FPDSCR_NS located at `0xE002EF3C`. The location `0xE002EF3C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

Field descriptions

The FPDSCR bit assignments are:



Bits [31:27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision. Default value for FPSCR.AHP.

This bit resets to zero on a Warm reset.

DN, bit [25]

Default NaN. Default value for FPSCR.DN.

This bit resets to zero on a Warm reset.

FZ, bit [24]

Flush-to-zero. Default value for FPSCR.FZ.

This bit resets to zero on a Warm reset.

RMode, bits [23:22]

Rounding mode. Default value for FPSCR.RMode.

This field resets to zero on a Warm reset.

Bits [21:0]

Reserved, RES0.

D1.2.83 FPSCR, Floating-point Status and Control Register

The FPSCR characteristics are:

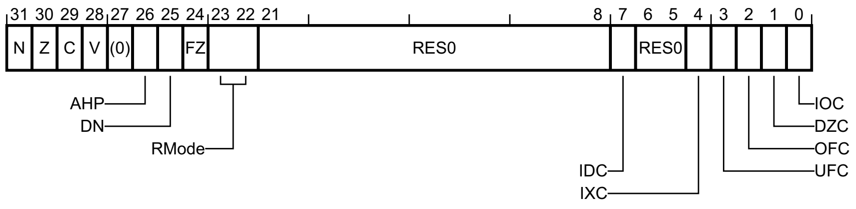
Purpose	Provides control of the floating-point system.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	Present only if the Floating-point Extension is implemented. This register is RES0 if the Floating-point Extension is not implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Preface

Writes to FPSCR can have side-effects on various aspects of processor operation. All of these side-effects are synchronous to FPSCR write. This means that they are guaranteed not to be visible to earlier instructions in the execution stream, and they are guaranteed to be visible to later instructions in the execution stream.

Field descriptions

The FPSCR bit assignments are:



N, bit [31]

Negative condition flag. When updated by a VCMP instruction, this bit indicates whether the result was less than.

The possible values of this bit are:

0	Compare result was not less than.
1	Compare result was less than.

See VCMP for details.

This bit resets to an UNKNOWN value on a Warm reset.

Z, bit [30]

Zero condition flag. When updated by a VCMP instruction, this bit indicates whether the result was equal.

The possible values of this bit are:

0	Compare result was not equal.
1	Compare result was equal.

See VCMP for details.

This bit resets to an UNKNOWN value on a Warm reset.

C, bit [29]

Carry condition flag. When updated by a VCMP instruction, this bit indicates whether the result was not less than.

The possible values of this bit are:

0	Compare result was less than.

1	Compare result was not less than.

See VCMP for details.

This bit resets to an UNKNOWN value on a Warm reset.

V, bit [28]

Overflow condition flag. When updated by a VCMP instruction, this bit indicates whether the result was unordered.

The possible values of this bit are:

0	Compare result was not unordered.
1	Compare result was unordered.

See VCMP for details.

This bit resets to an UNKNOWN value on a Warm reset.

Bit [27]

Reserved, RES0.

AHP, bit [26]

Alternative half-precision control bit. This bit controls how the PE interprets 16-bit floating-point values.

The possible values of this bit are:

0	IEEE half-precision format selected.
1	Alternative half-precision format selected.

This bit resets to an UNKNOWN value on a Warm reset.

DN, bit [25]

Default NaN mode control bit. This bit determines whether floating-point operations propagate NaNs or use the Default NaN.

The possible values of this bit are:

0	NaN operands propagate through to the output of a floating-point operation.
1	Any operation involving one of more NaNs returns the Default NaN.

This bit resets to an UNKNOWN value on a Warm reset.

FZ, bit [24]

Flush-to-zero mode control. This bit determines whether denormal floating-point values are treated as though zero.

The possible values of this bit are:

0	Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE754 standard.
1	Flush-to-zero mode enabled.

This bit resets to an UNKNOWN value on a Warm reset.

RMode, bits [23:22]

Rounding mode control field. This field determines what rounding mode is applied to floating-point operations.

The possible values of this field are:

`0b00`	Round to Nearest (RN) mode.
`0b01`	Round towards Plus Infinity (RP) mode.
`0b10`	Round towards Minus Infinity (RM) mode.
`0b11`	Round towards Zero (RZ) mode.

This field resets to an UNKNOWN value on a Warm reset.

Bits [21:8]

Reserved, RES0.

IDC, bit [7]

Input Denormal cumulative exception bit. This sticky flag records whether a floating-point input denormal exception has been detected since last cleared.

The possible values of this bit are:

0	Input Denormal exception has not occurred since 0 was last written to this bit.
1	Input Denormal exception has occurred since 0 was last written to this bit.

This bit resets to an UNKNOWN value on a Warm reset.

Bits [6:5]

Reserved, RES0.

IXC, bit [4]

Inexact cumulative exception bit. This sticky flag records whether a floating-point inexact exception has been detected since last cleared.

The possible values of this bit are:

0	Inexact exception has not occurred since 0 was last written to this bit.
1	Inexact exception has occurred since 0 was last written to this bit.

This bit resets to an UNKNOWN value on a Warm reset.

UFC, bit [3]

Underflow cumulative exception bit. This sticky flag records whether a floating-point Underflow exception has been detected since last cleared.

The possible values of this bit are:

0	Underflow exception has not occurred since 0 was last written to this bit.
1	Underflow exception has occurred since 0 was last written to this bit.

OFC, bit [2]

Overflow cumulative exception bit. This sticky flag records whether a floating-point overflow exception has been detected since last cleared.

The possible values of this bit are:

0	Overflow exception has not occurred since 0 was last written to this bit.
1	Overflow exception has occurred since 0 was last written to this bit.

This bit resets to an UNKNOWN value on a Warm reset.

DZC, bit [1]

Divide by Zero cumulative exception bit. This sticky flag records whether a floating-point divide by zero exception has been detected since last cleared.

The possible values of this bit are:

0	Division by Zero exception has not occurred since 0 was last written to this bit.
1	Division by Zero exception has occurred since 0 was last written to this bit.

This bit resets to an UNKNOWN value on a Warm reset.

IOC, bit [0]

Invalid Operation cumulative exception bit. This sticky flag records whether a floating-point invalid operation exception has been detected since last cleared.

The possible values of this bit are:

0	Invalid Operation exception has not occurred since 0 was last written to this bit.

1	Invalid Operation exception has occurred since 0 was last written to this bit.

This bit resets to an UNKNOWN value on a Warm reset.

D1.2.84 FP_CIDR0, FP Component Identification Register 0

The FP_CIDR0 characteristics are:

Purpose Provides CoreSight discovery information for the FP.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

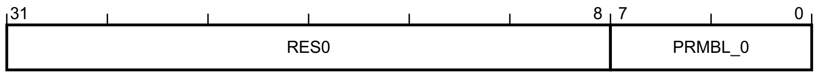
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FF0`.

This register is not banked between Security states.

Field descriptions

The FP_CIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

D1.2.85 FP_CIDR1, FP Component Identification Register 1

The FP_CIDR1 characteristics are:

Purpose Provides CoreSight discovery information for the FP.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

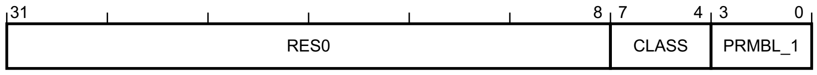
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `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`.

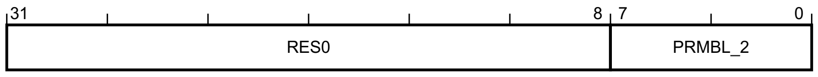
D1.2.86 FP_CIDR2, FP Component Identification Register 2

The FP_CIDR2 characteristics are:

Purpose	Provides CoreSight discovery information for the FP.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `0xE0002FF8`. This register is not banked between Security states.

Field descriptions

The FP_CIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

D1.2.87 FP_CIDR3, FP Component Identification Register 3

The FP_CIDR3 characteristics are:

Purpose Provides CoreSight discovery information for the FP.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

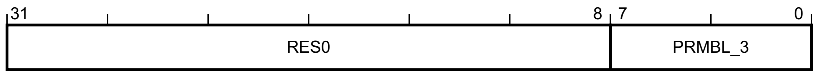
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FFC`.

This register is not banked between Security states.

Field descriptions

The FP_CIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

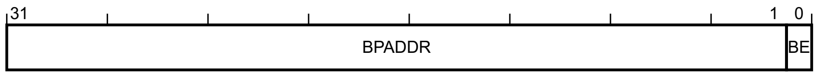
D1.2.88 FP_COMPn, Flash Patch Comparator Register, n = 0 - 125

The FP_COMP{0..125} characteristics are:

Purpose	Holds an address for comparison.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read/write register located at `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 write zero to the whole register.

This bit resets to zero on a Cold reset.

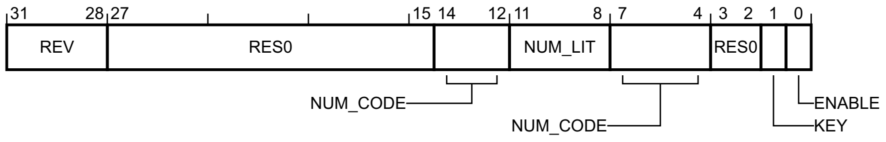
D1.2.89 FP_CTRL, Flash Patch Control Register

The FP_CTRL characteristics are:

Purpose	Provides FPB implementation information, and the global enable for the FPB unit.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
	If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the FPB is implemented.
	This register is RES0 if the FPB is not implemented.
Attributes	32-bit read/write register located at `0xE0002000`.
	This register is not banked between Security states.

Field descriptions

The FP_CTRL bit assignments are:



REV, bits [31:28]

Revision. Flash Patch and Breakpoint Unit architecture revision.

The possible values of this field are:

`0b0001` Flash Patch Breakpoint version 2 implemented.

All other values are reserved.

This field is read-only.

This field reads as `0b0001`.

Bits [27:15]

Reserved, RES0.

NUM_CODE, bits [14:12,7:4]

Number of implemented code comparators. Indicates the number of implemented instruction address comparators. Zero indicates no Instruction Address comparators are implemented. The Instruction Address comparators are numbered from 0 to NUM_CODE - 1.

This field is read-only.

This field reads as an IMPLEMENTATION DEFINED value.

NUM_LIT, bits [11:8]

Number of literal comparators. This field is RAZ/WI. Remapping is not supported in Armv8-M.

Bits [3:2]

Reserved, RES0.

KEY, bit [1]

FP_CTRL write-enable key. Writes to the FP_CTRL are ignored unless KEY is concurrently written to one.

The possible values of this bit are:

0	Concurrent write to FP_CTRL ignored.
1	Concurrent write to FP_CTRL permitted.

This bit reads-as-zero.

ENABLE, bit [0]

Flash Patch global enable. Enables the FPB.

The possible values of this bit are:

0	All FPB functionality disabled.
1	FPB enabled.

This bit resets to zero on a Cold reset.

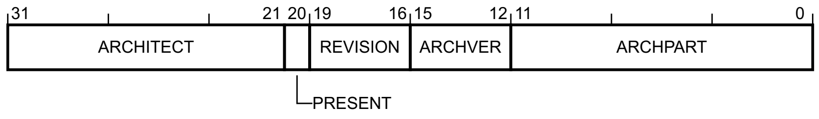
D1.2.90 FP_DEVARCH, FPB Device Architecture Register

The FP_DEVARCH characteristics are:

Purpose	Provides CoreSight discovery information for the FPB.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `0xE0002FBC`. This register is not banked between Security states.

Field descriptions

The FP_DEVARCH bit assignments are:



ARCHITECT, bits [31:21]

Architect. Defines the architect of the component. Bits [31:28] are the JEP106 continuation code (JEP106 bank ID, minus 1) and bits [27:21] are the JEP106 ID code.

The possible values of this field are:

`0x23B`	JEP106 continuation code `0x4`, ID code `0x3B`. Arm Limited.

Other values are defined by the JEDEC JEP106 standard.

This field reads as `0x23B`.

PRESENT, bit [20]

DEVARCH Present. Defines that the DEVARCH register is present.

The possible values of this bit are:

1	DEVARCH information present.

This bit reads as one.

REVISION, bits [19:16]

Revision. Defines the architecture revision of the component.

The possible values of this field are:

`0b0000`	FPB architecture v2.0.

This field reads as `0b0000`.

ARCHVER, bits [15:12]

Architecture Version. Defines the architecture version of the component.

The possible values of this field are:

`0b0001`	FPB architecture v2.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHVER is ARCHID[15:12].

This field reads as `0b0001`.

ARCHPART, bits [11:0]

Architecture Part. Defines the architecture of the component.

The possible values of this field are:

`0xA03`	FPB architecture.

ARCHVER and ARCHPART are also defined as a single field, ARCHID, so that ARCHPART is ARCHID[11:0].

This field reads as `0xA03`.

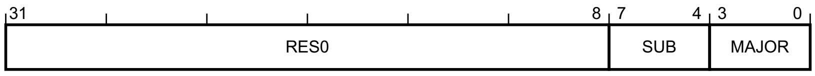
D1.2.91 FP_DEVTYPE, FPB Device Type Register

The FP_DEVTYPE characteristics are:

Purpose	Provides CoreSight discovery information for the FPB.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `0xE0002FCC`. This register is not banked between Security states.

Field descriptions

The FP_DEVTYPE bit assignments are:



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Sub-type. Component sub-type.

The possible values of this field are:

`0x0`	Other.

This field reads as `0b0000`.

MAJOR, bits [3:0]

Major type. Component major type.

The possible values of this field are:

`0x0`	Miscellaneous.

This field reads as `0b0000`.

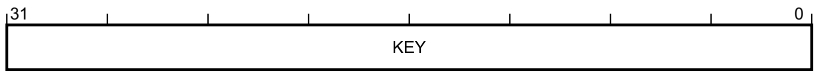
D1.2.92 FP_LAR, FPB Software Lock Access Register

The FP_LAR characteristics are:

Purpose	Provides CoreSight Software Lock control for the FPB, see the *ARM® CoreSight™ Architecture Specification* for details.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. This register is RAZ/WI if accessed via the debugger.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented. Present only if the optional Software Lock is implemented. This register is RAZ/WI if the Software Lock is not implemented.
Attributes	32-bit write-only register located at `0xE0002FB0`. This register is not banked between Security states.

Field descriptions

The FP_LAR bit assignments are:



KEY, bits [31:0]

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

D1.2.93 FP_LSR, FPB Software Lock Status Register

The FP_LSR characteristics are:

Purpose
Provides CoreSight Software Lock status information for the FPB, see the *ARM® CoreSight™ Architecture Specification* for details.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is RAZ/WI if accessed via the debugger.

Configurations
Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

This register is RES0 if the FPB is not implemented.

Present only if the optional Software Lock is implemented.

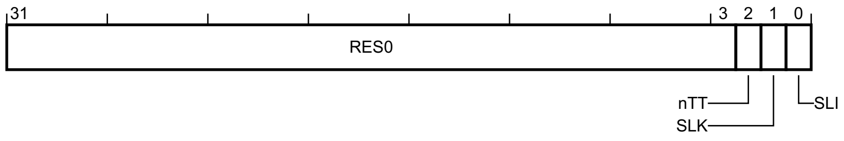
This register is RAZ/WI if the Software Lock is not implemented.

Attributes
32-bit read-only register located at `0xE0002FB4`.

This register is not banked between Security states.

Field descriptions

The FP_LSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as zero.

SLK, bit [1]

Software Lock status. See the *ARM® CoreSight™ Architecture Specification*.

The possible values of this bit are:

0	Lock clear. Software writes are permitted to this component's registers.
1	Lock set. Software writes to this component's registers are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Cold reset.

SLI, bit [0]

Software Lock implemented. See the *ARM® CoreSight™ Architecture Specification*.

The possible values of this bit are:

0	Software Lock not implemented or debugger access.
1	Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

D1.2.94 FP_PIDR0, FP Peripheral Identification Register 0

The FP_PIDR0 characteristics are:

Purpose Provides CoreSight discovery information for the FPB.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

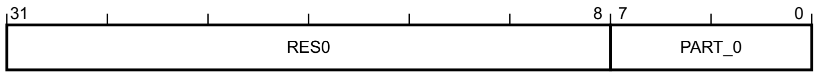
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FE0`.

This register is not banked between Security states.

Field descriptions

The FP_PIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number bits [7:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

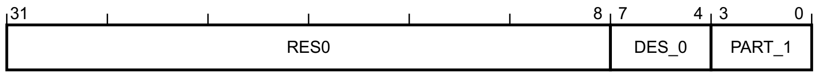
D1.2.95 FP_PIDR1, FP Peripheral Identification Register 1

The FP_PIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the FPB.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `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.

D1.2.96 FP_PIDR2, FP Peripheral Identification Register 2

The FP_PIDR2 characteristics are:

Purpose Provides CoreSight discovery information for the FP.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

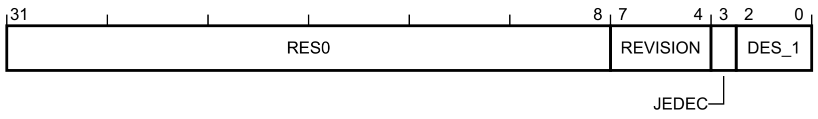
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FE8`.

This register is not banked between Security states.

Field descriptions

The FP_PIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component revision. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

JEDEC, bit [3]

JEDEC assignee value is used. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as one.

DES_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

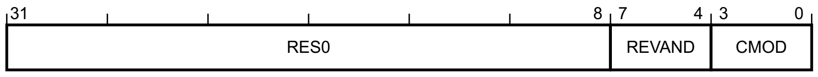
D1.2.97 FP_PIDR3, FP Peripheral Identification Register 3

The FP_PIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the FPB.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `0xE0002FEC`. This register is not banked between Security states.

Field descriptions

The FP_PIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

RevAnd. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

CMOD, bits [3:0]

Customer Modified. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.98 FP_PIDR4, FP Peripheral Identification Register 4

The FP_PIDR4 characteristics are:

Purpose Provides CoreSight discovery information for the FPB.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

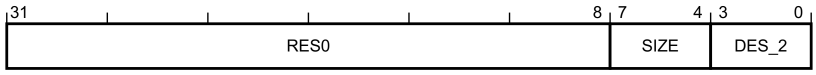
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FD0`.

This register is not banked between Security states.

Field descriptions

The FP_PIDR4 bit assignments are:



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

4KB count. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as zero.

DES_2, bits [3:0]

JEP106 continuation code. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

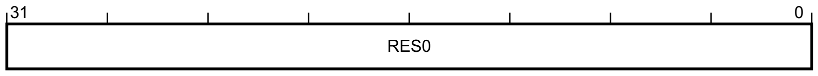
D1.2.99 FP_PIDR5, FP Peripheral Identification Register 5

The FP_PIDR5 characteristics are:

Purpose	Provides CoreSight discovery information for the FPB.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the FPB is implemented. This register is RES0 if the FPB is not implemented.
Attributes	32-bit read-only register located at `0xE0002FD4`. This register is not banked between Security states.

Field descriptions

The FP_PIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.100 FP_PIDR6, FP Peripheral Identification Register 6

The FP_PIDR6 characteristics are:

Purpose Provides CoreSight discovery information for the FPB.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

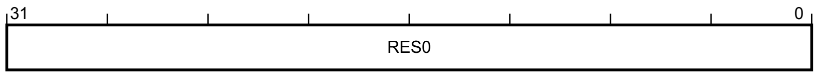
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FD8`.

This register is not banked between Security states.

Field descriptions

The FP_PIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.101 FP_PIDR7, FP Peripheral Identification Register 7

The FP_PIDR7 characteristics are:

Purpose Provides CoreSight discovery information for the FPB.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the FPB is implemented.

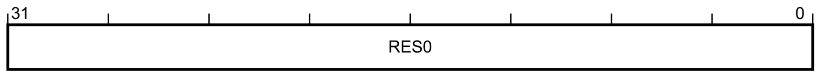
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002FDC`.

This register is not banked between Security states.

Field descriptions

The FP_PIDR7 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.102 FP_REMAP, Flash Patch Remap Register

The FP_REMAP characteristics are:

Purpose Indicates whether the implementation supports Flash Patch remap and, if it does, holds the target address for remap.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if the FPB is implemented.

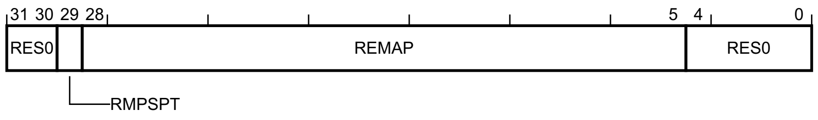
This register is RES0 if the FPB is not implemented.

Attributes 32-bit read-only register located at `0xE0002004`.

This register is not banked between Security states.

Field descriptions

The FP_REMAP bit assignments are:



Bits [31:30]

Reserved, RES0.

RMPSPT, bit [29]

Remap supported. This field is RAZ. Remapping is not supported in Armv8-M.

REMAP, bits [28:5]

Remap address.

Reserved, RES0.

Bits [4:0]

Reserved, RES0.

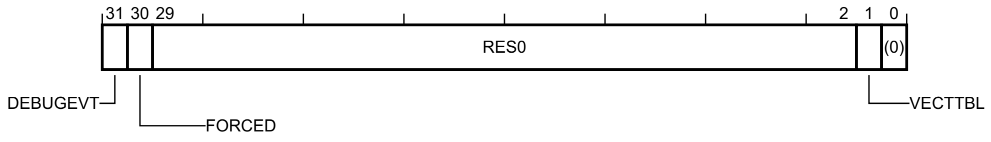
D1.2.103 HFSR, HardFault Status Register

The HFSR characteristics are:

Purpose	Shows the cause of any HardFaults.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write-one-to-clear register located at `0xE000ED2C`. Secure software can access the Non-secure version of this register via HFSR_NS located at `0xE002ED2C`. The location `0xE002ED2C` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The HFSR bit assignments are:



DEBUGEVT, bit [31]

Debug event. Indicates when a debug event has occurred.

The possible values of this bit are:

0	No debug event has occurred.
1	Debug event has occurred. The Debug Fault Status Register has been updated.

The PE sets this bit to 1 only when Halting debug is disabled and a debug event occurs. 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.

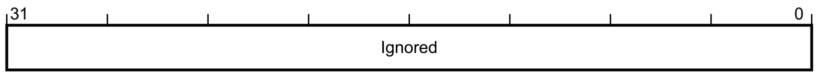
D1.2.104 ICIALLU, Instruction Cache Invalidate All to PoU

The ICIALLU characteristics are:

Purpose	Invalidate all instruction caches to PoU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit write-only register located at `0xE000EF50`. Secure software can access the Non-secure version of this register via ICIALLU_NS located at `0xE002EF50`. The location `0xE002EF50` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ICIALLU bit assignments are:



Ignored, bits [31:0]

The value written to this field is ignored. Ignored.

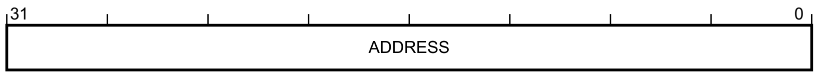
D1.2.105 ICIMVAU, Instruction Cache line Invalidate by Address to PoU

The ICIMVAU characteristics are:

Purpose	Invalidate instruction cache line by address to PoU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit write-only register located at `0xE000EF58`. Secure software can access the Non-secure version of this register via ICIMVAU_NS located at `0xE002EF58`. The location `0xE002EF58` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ICIMVAU bit assignments are:



ADDRESS, bits [31:0]

Address. Writing to this field initiates the maintenance operation for the address written.

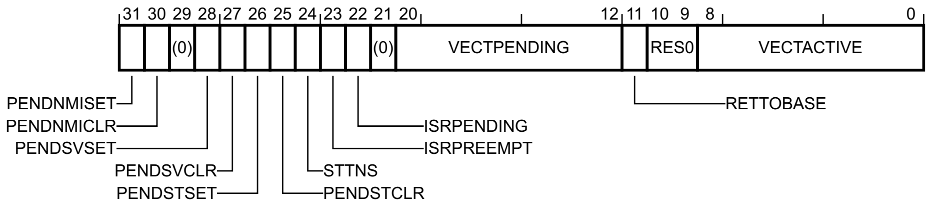
D1.2.106 ICSR, Interrupt Control and State Register

The ICSR characteristics are:

Purpose	Controls and provides status information for NMI, PendSV, SysTick and interrupts.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `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 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.

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 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.

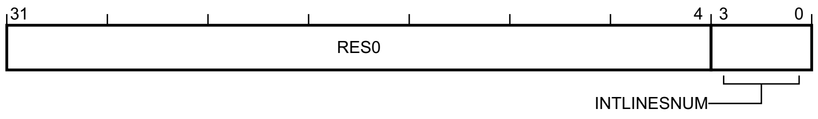
D1.2.107 ICTR, Interrupt Controller Type Register

The ICTR characteristics are:

Purpose	Provides information about the interrupt controller.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000E004`. Secure software can access the Non-secure version of this register via ICTR_NS located at `0xE002E004`. The location `0xE002E004` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ICTR bit assignments are:



Bits [31:4]

Reserved, RES0.

INTLINESNUM, bits [3:0]

Interrupt line set number. Indicates the number of the highest implemented register in each of the NVIC control register sets, or in the case of NVIC_IPR*n*, 4xINTLINESNUM.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.108 ID_AFR0, Auxiliary Feature Register 0

The ID_AFR0 characteristics are:

Purpose Provides information about the IMPLEMENTATION DEFINED features of the PE.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

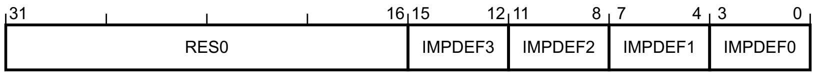
Attributes 32-bit read-only register located at `0xE000ED4C`.

Secure software can access the Non-secure version of this register via ID_AFR0_NS located at `0xE002ED4C`. The location `0xE002ED4C` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The ID_AFR0 bit assignments are:



Bits [31:16]

Reserved, RES0.

IMPDEF*m*, bits [4*m*+3:4*m*], for *m* = 0 to 3

IMPLEMENTATION DEFINED. IMPLEMENTATION DEFINED meaning.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.109 ID_DFR0, Debug Feature Register 0

The ID_DFR0 characteristics are:

Purpose	Provides top level information about the debug system.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED48`. Secure software can access the Non-secure version of this register via ID_DFR0_NS located at `0xE002ED48`. The location `0xE002ED48` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

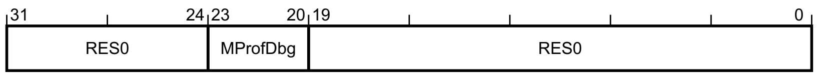
Preface

If Halting debug is not implemented this register reads as `0x00000000`.

If Halting debug is implemented this register reads as `0x00200000`.

Field descriptions

The ID_DFR0 bit assignments are:



Bits [31:24]

Reserved, RES0.

MProfDbg, bits [23:20]

M-profile debug. Indicates the supported M-profile debug architecture.

The possible values of this field are:

`0b0000`	Halting debug is not implemented.
`0b0010`	Armv8-M Debug architecture.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [19:0]

Reserved, RES0.

D1.2.110 ID_ISAR0, Instruction Set Attribute Register 0

The ID_ISAR0 characteristics are:

Purpose	Provides information about the instruction set implemented by the PE.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED60`. Secure software can access the Non-secure version of this register via ID_ISAR0_NS located at `0xE002ED60`. The location `0xE002ED60` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

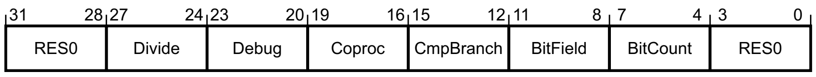
Preface

If coprocessors excluding the Floating-point Extension are not supported this register reads as `0x01101110`.

If coprocessors excluding the Floating-point Extension are supported this register reads as `0x01141110`.

Field descriptions

The ID_ISAR0 bit assignments are:



Bits [31:28]

Reserved, RES0.

Divide, bits [27:24]

Divide. Indicates the supported Divide instructions.

The possible values of this field are:

`0b0001` Supports SDIV and UDIV instructions.

All other values are reserved.

This field reads as `0b0001`.

Debug, bits [23:20]

Debug. Indicates the implemented Debug instructions.

The possible values of this field are:

`0b0001` Supports BKPT instruction.

All other values are reserved.

This field reads as `0b0001`.

Coproc, bits [19:16]

Coprocessor. Indicates the supported coprocessor instructions.

The possible values of this field are:

`0b0000` No coprocessor instructions support other than FPU.

`0b0100` Coprocessor instructions supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

CmpBranch, bits [15:12]

Compare and branch. Indicates the supported combined Compare and Branch instructions.

The possible values of this field are:

`0b0001`	Supports CBNZ and CBZ instructions.

All other values are reserved.

This field reads as `0b0001`.

BitField, bits [11:8]

Bit field. Indicates the supported bit field instructions.

The possible values of this field are:

`0b0001`	BFC, BFI, SBFX, and UBFX supported.

All other values are reserved.

This field reads as `0b0001`.

BitCount, bits [7:4]

Bit count. Indicates the supported bit count instructions.

The possible values of this field are:

`0b0001`	CLZ supported.

All other values are reserved.

This field reads as `0b0001`.

Bits [3:0]

Reserved, RES0.

D1.2.111 ID_ISAR1, Instruction Set Attribute Register 1

The ID_ISAR1 characteristics are:

Purpose	Provides information about the instruction set implemented by the PE.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED64`. Secure software can access the Non-secure version of this register via ID_ISAR1_NS located at `0xE002ED64`. The location `0xE002ED64` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

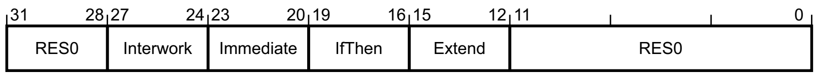
Preface

If the DSP Extension is not implemented, this register reads as `0x02211000`.

If the DSP Extension is implemented, this register reads as `0x02212000`.

Field descriptions

The ID_ISAR1 bit assignments are:



Bits [31:28]

Reserved, RES0.

Interwork, bits [27:24]

Interworking. Indicates the implemented interworking instructions.

The possible values of this field are:

`0b0010`	`BLX`, `BX`, and loads to PC interwork.

All other values are reserved.

This field reads as `0b0010`.

Immediate, bits [23:20]

Immediate. Indicates the implemented for data-processing instructions with long immediates.

The possible values of this field are:

`0b0010`	`ADDW`, `MOVW`, `MOVT`, and `SUBW` supported.

All other values are reserved.

This field reads as `0b0010`.

IfThen, bits [19:16]

If-Then. Indicates the implemented If-Then instructions.

The possible values of this field are:

`0b0001`	IT instruction supported.

All other values are reserved.

This field reads as `0b0001`.

Extend, bits [15:12]

Extend. Indicates the implemented Extend instructions.

The possible values of this field are:

`0b0001`	`SXTB`, `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.

D1.2.112 ID_ISAR2, Instruction Set Attribute Register 2

The ID_ISAR2 characteristics are:

Purpose Provides information about the instruction set implemented by the PE.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

Attributes 32-bit read-only register located at `0xE000ED68`.

Secure software can access the Non-secure version of this register via ID_ISAR2_NS located at `0xE002ED68`. The location `0xE002ED68` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Preface

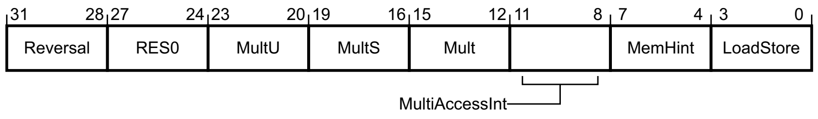
With bits [11:8] masked, if the DSP Extension is not implemented, this register reads as `0x20112032`.

With bits[11:8] masked, if the DSP Extension is implemented, this register reads as `0x20232032`.

The value of bits [11:8] is determined by whether the PE implements restartable or continuable multi-access instructions.

Field descriptions

The ID_ISAR2 bit assignments are:



Reversal, bits [31:28]

Reversal. Indicates the implemented Reversal instructions.

The possible values of this field are:

`0b0010` `REV`, `REV16`, `REVSH` and `RBIT` instructions supported.

All other values are reserved.

This field reads as `0b0010`.

Bits [27:24]

Reserved, RES0.

MultU, bits [23:20]

Multiply unsigned. Indicates the implemented advanced unsigned Multiply instructions.

The possible values of this field are:

`0b0001` `UMULL` and `UMLAL`.

`0b0010` Adds `UMAAL`, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

MultS, bits [19:16]

Multiply signed. Indicates the implemented advanced signed Multiply instructions.

The possible values of this field are:

`0b0001`	`SMULL` and `SMLAL`.
`0b0011`	Adds all saturating and DSP signed multiplies, DSP Extension only.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

Mult, bits [15:12]

Multiplies. Indicates the implemented additional Multiply instructions.

The possible values of this field are:

`0b0010`	`MUL`, `MLA`, and `MLS`.

All other values are reserved.

This field reads as `0b0010`.

MultiAccessInt, bits [11:8]

Multi-access instructions. Indicates the support for interruptible multi-access instructions.

The possible values of this field are:

`0b0000`	No support. `LDM` and `STM` instructions are not interruptible.
`0b0001`	`LDM` and `STM` instructions are restartable.
`0b0010`	`LDM` and `STM` instructions are continuable.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

MemHint, bits [7:4]

Memory hints. Indicates the implemented Memory hint instructions.

The possible values of this field are:

`0b0011`	`PLI` and `PLD` instructions implemented.

All other values are reserved.

This field reads as `0b0011`.

LoadStore, bits [3:0]

Load/store. Indicates the implemented additional load/store instructions.

The possible values of this field are:

`0b0010`	Supports load-acquire, store-release, and exclusive load and store instructions.

All other values are reserved.

This field reads as `0b0010`.

D1.2.113 ID_ISAR3, Instruction Set Attribute Register 3

The ID_ISAR3 characteristics are:

Purpose	Provides information about the instruction set implemented by the PE.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED6C`. Secure software can access the Non-secure version of this register via ID_ISAR3_NS located at `0xE002ED6C`. The location `0xE002ED6C` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

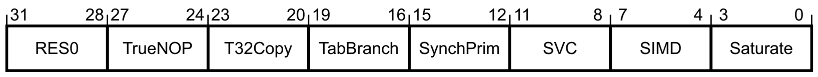
Preface

If the DSP Extension is not implemented, this register reads as `0x01111110`.

If the DSP Extension is implemented, this register reads as `0x01111131`.

Field descriptions

The ID_ISAR3 bit assignments are:



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.

D1.2.114 ID_ISAR4, Instruction Set Attribute Register 4

The ID_ISAR4 characteristics are:

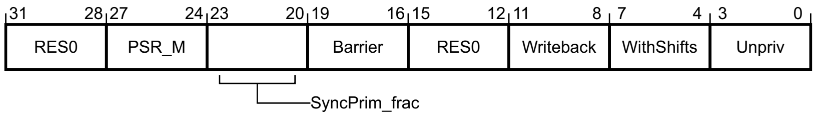
Purpose	Provides information about the instruction set implemented by the PE.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED70`. Secure software can access the Non-secure version of this register via ID_ISAR4_NS located at `0xE002ED70`. The location `0xE002ED70` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Preface

This register reads as `0x01310131`.

Field descriptions

The ID_ISAR4 bit assignments are:



Bits [31:28]

Reserved, RES0.

PSR_M, bits [27:24]

Program Status Registers M. Indicates the implemented M profile instructions to modify the PSRs.

The possible values of this field are:

`0b0001`	M profile forms of `CPS`, `MRS`, and `MSR` implemented.

All other values are reserved.

This field reads as `0b0001`.

SyncPrim_frac, bits [23:20]

Synchronization primitives fractional. Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented synchronization primitive instructions.

The possible values of this field are:

`0b0011`	`LDREX`, `STREX`, `CLREX`, `LDREXB`, `LDREXH`, `STREXB`, and `STREXH` implemented.

All other values are reserved.

This field reads as `0b0011`.

Barrier, bits [19:16]

Barrier. Indicates the implemented Barrier instructions.

The possible values of this field are:

`0b0001`	`DMB`, `DSB`, and `ISB` barrier instructions implemented.

All other values are reserved.

This field reads as `0b0001`.

Bits [15:12]

Reserved, RES0.

Writeback, bits [11:8]

Writeback. Indicates the support for writeback addressing modes.

The possible values of this field are:

`0b0001`	All writeback addressing modes supported.

All other values are reserved.

This field reads as `0b0001`.

WithShifts, bits [7:4]

With shifts. Indicates the support for write-back addressing modes.

The possible values of this field are:

`0b0011`	Support for constant shifts on load/store and other instructions.

All other values are reserved.

This field reads as `0b0011`.

Unpriv, bits [3:0]

Unprivileged. Indicates the implemented unprivileged instructions.

The possible values of this field are:

`0b0010`	`LDRBT`, `LDRHT`, `LDRSBT`, `LDRSHT`, `LDRT`, `STRBT`, `STRHT`, and `STRT` implemented.

All other values are reserved.

This field reads as `0b0010`.

D1.2.115 ID_ISAR5, Instruction Set Attribute Register 5

The ID_ISAR5 characteristics are:

Purpose Provides information about the instruction set implemented by the PE.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

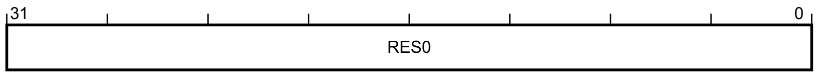
Attributes 32-bit read-only register located at `0xE000ED74`.

Secure software can access the Non-secure version of this register via ID_ISAR5_NS located at `0xE002ED74`. The location `0xE002ED74` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The ID_ISAR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

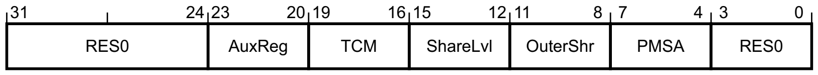
D1.2.116 ID_MMFR0, Memory Model Feature Register 0

The ID_MMFR0 characteristics are:

Purpose	Provides information about the implemented memory model and memory management support.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED50`. Secure software can access the Non-secure version of this register via ID_MMFR0_NS located at `0xE002ED50`. The location `0xE002ED50` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ID_MMFR0 bit assignments are:



Bits [31:24]

Reserved, RES0.

AuxReg, bits [23:20]

Auxiliary Registers. Indicates support for Auxiliary Control Registers.

The possible values of this field are:

`0b0000`	No Auxiliary Control Registers.
`0b0001`	Auxiliary Control Registers supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

TCM, bits [19:16]

Tightly Coupled Memories. Indicates support for Tightly Coupled Memories (TCMs).

The possible values of this field are:

`0b0000`	None supported.
`0b0001`	TCMs supported with IMPLEMENTATION DEFINED control.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

ShareLvl, bits [15:12]

Shareability Levels. Indicates the number of Shareability levels implemented.

The possible values of this field are:

`0b0000`	One level of Shareability implemented.
`0b0001`	Two levels of Shareability implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

OuterShr, bits [11:8]

Outermost Shareability. Indicates the outermost Shareability domain implemented.

The possible values of this field are:

`0b0000`	Implemented as Non-cacheable.
`0b0001`	Implemented with hardware coherency support.
`0b1111`	Shareability ignored.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

PMSA, bits [7:4]

Protected memory system architecture. Indicates support for the protected memory system architecture (PMSA).

The possible values of this field are:

`0b0100`	Supports PMSAv8.

All other values are reserved.

This field reads as `0b0100`.

Bits [3:0]

Reserved, RES0.

D1.2.117 ID_MMFR1, Memory Model Feature Register 1

The ID_MMFR1 characteristics are:

Purpose Provides information about the implemented memory model and memory management support.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

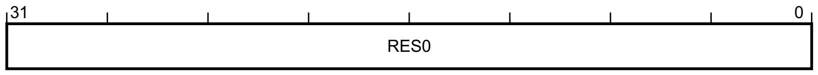
Attributes 32-bit read-only register located at `0xE000ED54`.

Secure software can access the Non-secure version of this register via ID_MMFR1_NS located at `0xE002ED54`. The location `0xE002ED54` is RES0 to software executing in Non-secure state and the debugger.

This register is not banked between Security states.

Field descriptions

The ID_MMFR1 bit assignments are:



Bits [31:0]

Reserved, RES0.

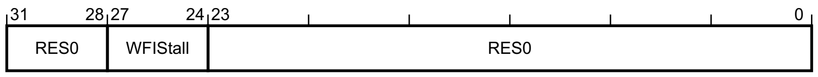
D1.2.118 ID_MMFR2, Memory Model Feature Register 2

The ID_MMFR2 characteristics are:

Purpose	Provides information about the implemented memory model and memory management support.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED58`. Secure software can access the Non-secure version of this register via ID_MMFR2_NS located at `0xE002ED58`. The location `0xE002ED58` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ID_MMFR2 bit assignments are:



Bits [31:28]

Reserved, RES0.

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.

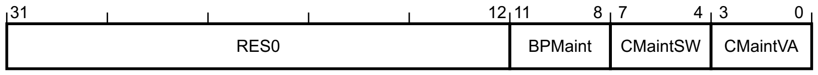
D1.2.119 ID_MMFR3, Memory Model Feature Register 3

The ID_MMFR3 characteristics are:

Purpose	Provides information about the implemented memory model and memory management support.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000ED5C`. Secure software can access the Non-secure version of this register via ID_MMFR3_NS located at `0xE002ED5C`. The location `0xE002ED5C` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Field descriptions

The ID_MMFR3 bit assignments are:



Bits [31:12]

Reserved, RES0.

BPMaint, bits [11:8]

Branch predictor maintenance. Indicates the supported branch predictor maintenance.

The possible values of this field are:

`0b0000`	None supported.
`0b0001`	Support for invalidate all of branch predictors.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

CMaintSW, bits [7:4]

Cache maintenance set/way. Indicates the supported cache maintenance operations by set/way.

The possible values of this field are:

`0b0000`	None supported.
`0b0001`	Maintenance by set/way operations supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

CMaintVA, bits [3:0]

Cache maintenance by address. Indicates the supported cache maintenance operations by address.

The possible values of this field are:

`0b0000`	None supported.
`0b0001`	Maintenance by address and instruction cache invalidate all supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.120 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.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

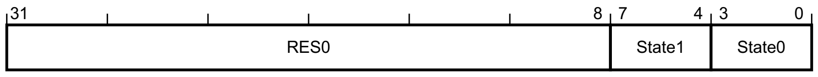
Attributes 32-bit read-only register located at `0xE000ED40`.

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:



Bits [31:8]

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`.

D1.2.121 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.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

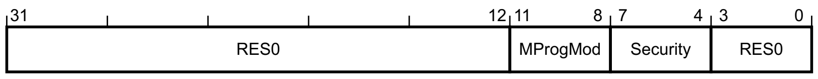
Attributes 32-bit read-only register located at `0xE000ED44`.

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.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [3:0]

Reserved, RES0.

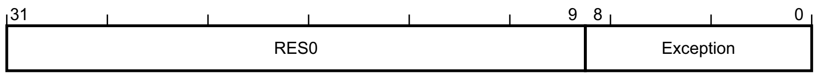
D1.2.122 IPSR, Interrupt Program Status Register

The IPSR characteristics are:

Purpose	Provides privileged access to the current exception number field.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The IPSR bit assignments are:



Bits [31:9]

Reserved, RES0.

Exception, bits [8:0]

Exception number. Holds the exception number of the currently-executing exception, or zero for Thread mode.

The possible values of this field are:

Zero	PE in Thread mode.
Non zero	PE in Handler mode in given exception number. On a function call from Secure state the value is set to 1 to ensure that the Non-secure state cannot determine which exception handler is executing.

This field resets to zero on a Warm reset.

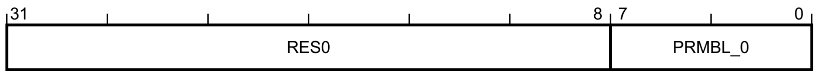
D1.2.123 ITM_CIDR0, ITM Component Identification Register 0

The ITM_CIDR0 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FF0`. This register is not banked between Security states.

Field descriptions

The ITM_CIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

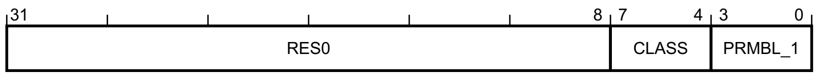
D1.2.124 ITM_CIDR1, ITM Component Identification Register 1

The ITM_CIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FF4`. This register is not banked between Security states.

Field descriptions

The ITM_CIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

CoreSight component class. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

PRMBL_1, bits [3:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

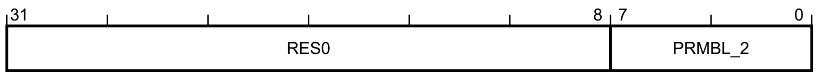
D1.2.125 ITM_CIDR2, ITM Component Identification Register 2

The ITM_CIDR2 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FF8`. This register is not banked between Security states.

Field descriptions

The ITM_CIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

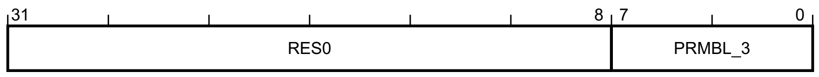
D1.2.126 ITM_CIDR3, ITM Component Identification Register 3

The ITM_CIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FFC`. This register is not banked between Security states.

Field descriptions

The ITM_CIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_3, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0xB1`.

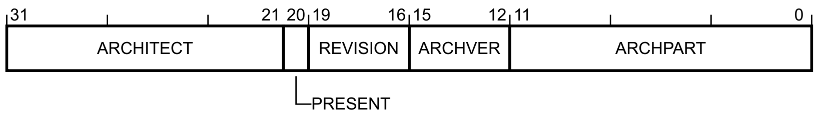
D1.2.127 ITM_DEVARCH, ITM Device Architecture Register

The ITM_DEVARCH characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. 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`.

D1.2.128 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 BusFault.

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.

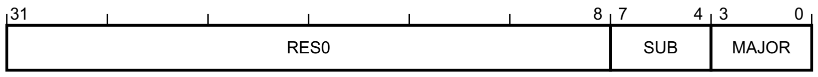
If the Main Extension is not implemented then the ITM is not implemented.

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.

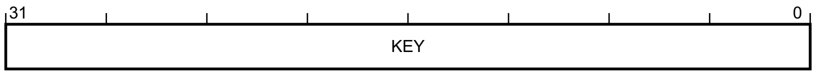
D1.2.129 ITM_LAR, ITM Software Lock Access Register

The ITM_LAR characteristics are:

Purpose	Provides CoreSight Software Lock control for the ITM, see the *ARM® CoreSight™ Architecture Specification* for details.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored.
	If the Main Extension is not implemented, unprivileged accesses generate a BusFault.
	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 this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

D1.2.130 ITM_LSR, ITM Software Lock Status Register

The ITM_LSR characteristics are:

Purpose Provides CoreSight Software Lock status information for the ITM, see the *ARM® CoreSight™ Architecture Specification* for details.

Usage constraints If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a BusFault.

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.

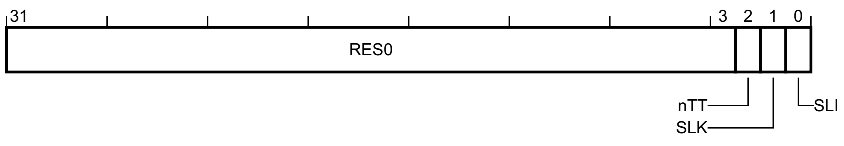
This register is RAZ/WI if the Software Lock is not implemented.

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 this component's registers.
1	Lock set. Software writes to this component's registers are ignored, and reads have no side-effects.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RES0.

This bit resets to one on a Warm reset.

SLI, bit [0]

Software Lock implemented. See the *ARM® CoreSight™ Architecture Specification*.

The possible values of this bit are:

0	Software Lock not implemented or debugger access.

1	Software Lock is implemented and software access.

For a debugger read of this register, or when the Software Lock is not implemented, this bit is RAZ.

This bit reads as an IMPLEMENTATION DEFINED value.

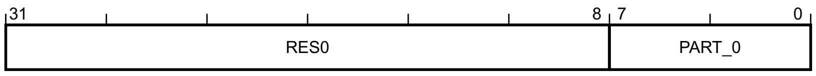
D1.2.131 ITM_PIDR0, ITM Peripheral Identification Register 0

The ITM_PIDR0 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FE0`. This register is not banked between Security states.

Field descriptions

The ITM_PIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PART_0, bits [7:0]

Part number bits [7:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

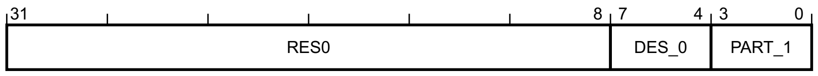
D1.2.132 ITM_PIDR1, ITM Peripheral Identification Register 1

The ITM_PIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FE4`. This register is not banked between Security states.

Field descriptions

The ITM_PIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

PART_1, bits [3:0]

Part number bits [11:8]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.133 ITM_PIDR2, ITM Peripheral Identification Register 2

The ITM_PIDR2 characteristics are:

Purpose Provides CoreSight discovery information for the ITM.

Usage constraints If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a BusFault.

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.

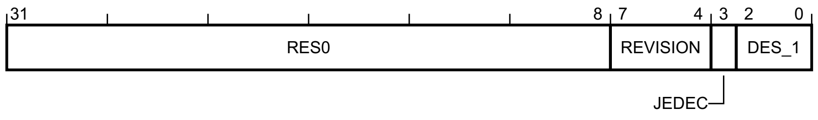
If the Main Extension is not implemented then the ITM is not implemented.

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.

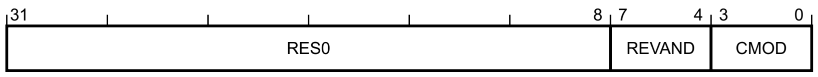
D1.2.134 ITM_PIDR3, ITM Peripheral Identification Register 3

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 BusFault. 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.

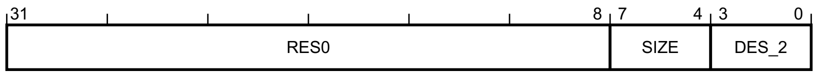
D1.2.135 ITM_PIDR4, ITM Peripheral Identification Register 4

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 BusFault. 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.

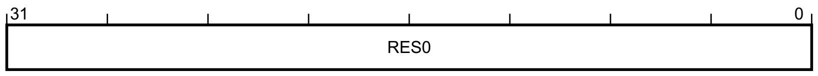
D1.2.136 ITM_PIDR5, ITM Peripheral Identification Register 5

The ITM_PIDR5 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.
	If the Main Extension is not implemented, unprivileged accesses generate a BusFault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented.
	This register is RES0 if CoreSight identification is not implemented.
	Present only if the ITM is implemented.
	This register is RES0 if the ITM is not implemented.
	If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FD4`.
	This register is not banked between Security states.

Field descriptions

The ITM_PIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

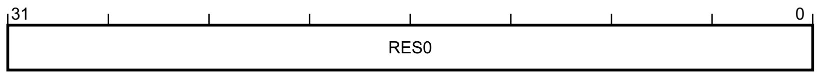
D1.2.137 ITM_PIDR6, ITM Peripheral Identification Register 6

The ITM_PIDR6 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read-only register located at `0xE0000FD8`. This register is not banked between Security states.

Field descriptions

The ITM_PIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

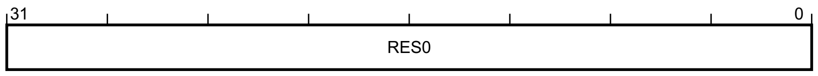
D1.2.138 ITM_PIDR7, ITM Peripheral Identification Register 7

The ITM_PIDR7 characteristics are:

Purpose	Provides CoreSight discovery information for the ITM.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. 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.

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

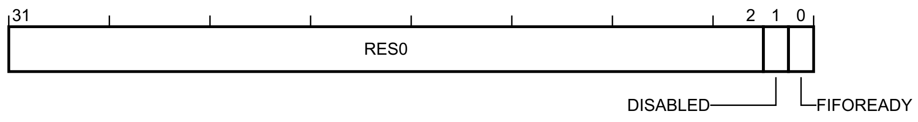
The ITM_STIM{0..255} characteristics are:

Purpose	Provides the interface for generating Instrumentation packets.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored if ITM_TPR.PRIVMASK[*n* DIV 8] is set to one. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. 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.

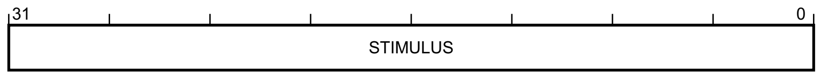
Field descriptions

The ITM_STIM{0..255} bit assignments are:

On a read:



On a write:



STIMULUS, bits [31:0], on a write

Stimulus data. Data to write to the stimulus port output buffer, for forwarding as an Instrumentation packet. The size of write access determines the type of Instrumentation packet generated.

Bits [31:2], on a read

Reserved, RES0.

DISABLED, bit [1], on a read

Disabled. Indicates whether the stimulus port is enabled or disabled.

The possible values of this bit are:

0	Stimulus port and ITM are enabled.
1	Stimulus port or ITM is disabled.

FIFOREADY, bit [0], on a read

FIFO ready. Indicates whether the stimulus port can accept data.

The possible values of this bit are:

0	Stimulus port cannot accept data.

1	Stimulus port can accept at least one word.

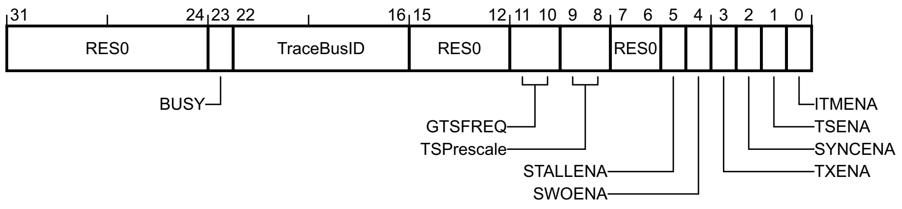
D1.2.140 ITM_TCR, ITM Trace Control Register

The ITM_TCR characteristics are:

Purpose	Configures and controls transfers through the ITM interface.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. 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.

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

The ITM_TER{0..7} characteristics are:

Purpose Provide an individual enable bit for each ITM_STIM register.

Usage constraints If the Main Extension is implemented, both privileged and unprivileged accesses are permitted.

If the Main Extension is not implemented, unprivileged accesses generate a BusFault.

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.

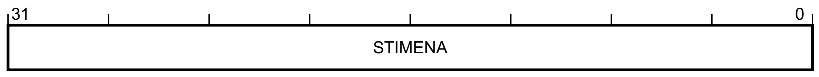
If the Main Extension is not implemented then the ITM is not implemented.

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.

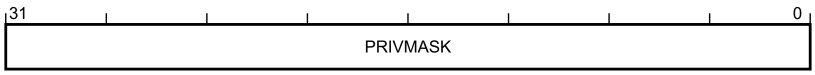
D1.2.142 ITM_TPR, ITM Trace Privilege Register

The ITM_TPR characteristics are:

Purpose	Controls which stimulus ports can be accessed by unprivileged code.
Usage constraints	If the Main Extension is implemented, both privileged and unprivileged accesses are permitted, but unprivileged writes are ignored. If the Main Extension is not implemented, unprivileged accesses generate a BusFault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the ITM is implemented. This register is RES0 if the ITM is not implemented. If the Main Extension is not implemented then the ITM is not implemented.
Attributes	32-bit read/write register located at `0xE0000E40`. This register is not banked between Security states.

Field descriptions

The ITM_TPR bit assignments are:



PRIVMASK, bits [31:0]

Privilege mask. For PRIVMASK[*m*], defines the access permissions of stimulus ports ITM_STIM<8*m*> to ITM_STIM<8*m*+7> inclusive.

The possible values of each bit are:

0	Unprivileged access permitted.
1	Privileged access only.

Bits corresponding to unimplemented stimulus ports are RAZ/WI.

This field resets to zero on a Cold reset.

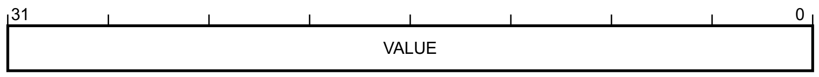
D1.2.143 LR, Link Register

The LR characteristics are:

Purpose	Exception and procedure call link register.
Usage constraints	Privileged and unprivileged access permitted.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The LR bit assignments are:



VALUE, bits [31:0]

Link register. 32-bit link register updated to hold a return address, FNC_RETURN or EXC_RETURN on a function call or exception entry. LR can be used as a general-purpose register.

This field resets to an UNKNOWN value on Warm reset when the Main Extension is not implemented.

This field resets to `0xFFFFFFFF` on a Warm reset if the Main Extension is implemented.

D1.2.144 MAIR_ATTR, Memory Attribute Indirection Register Attributes

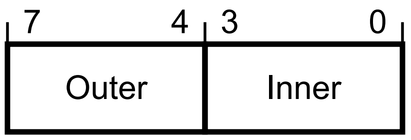
The MAIR_ATTR characteristics are:

Purpose	Defines the memory attribute encoding for use in the MPU_MAIR0 and MPU_MAIR1.
Usage constraints	None.
Configurations	All.
Attributes	8-bit payload.

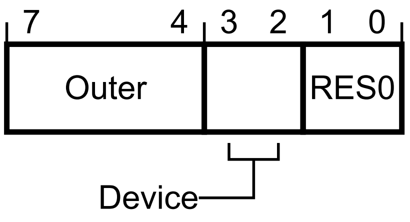
Field descriptions

The MAIR_ATTR bit assignments are:

When `Outer != 0b0000`:



When `Outer == 0b0000`:



Outer, bits [7:4]

Outer attributes. Specifies the Outer memory attributes.

The possible values of this field are:

`0b0000`	Device memory.
`0b00RW`	Normal memory, Outer Write-Through transient (RW!=`0b00`).
`0b0100`	Normal memory, Outer Non-cacheable.
`0b01RW`	Normal memory, Outer Write-Back Transient (RW!=`0b00`).
`0b10RW`	Normal memory, Outer Write-Through Non-transient.
`0b11RW`	Normal memory, Outer Write-Back Non-transient.

R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

Device, bits [3:2], when Outer == `0b0000`

Device attributes. Specifies the memory attributes for Device.

The possible values of this field are:

`0b00`	Device-nGnRnE.
`0b01`	Device-nGnRE.
`0b10`	Device-nGRE.
`0b11`	Device-GRE.

Bits [1:0], when Outer == `0b0000`

Reserved, RES0.

Inner, bits [3:0], when Outer != `0b0000`

Inner attributes. Specifies the Inner memory attributes.

The possible values of this field are:

`0b0000`	UNPREDICTABLE.
`0b00RW`	Normal memory, Inner Write-Through Transient (RW!=`0b00`).
`0b0100`	Normal memory, Inner Non-cacheable.
`0b01RW`	Normal memory, Inner Write-Back Transient (RW!=`0b00`).
`0b10RW`	Normal memory, Inner Write-Through Non-transient.
`0b11RW`	Normal memory, Inner Write-Back Non-transient.

R and W specify the inner read and write allocation policy: 0 = do not allocate, 1 = allocate.

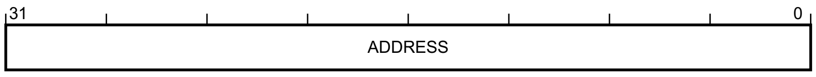
D1.2.145 MMFAR, MemManage Fault Address Register

The MMFAR characteristics are:

Purpose	Shows the address of the memory location that caused an MPU fault.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write register located at `0xE000ED34`. Secure software can access the Non-secure version of this register via MMFAR_NS located at `0xE002ED34`. The location `0xE002ED34` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Field descriptions

The MMFAR bit assignments are:



ADDRESS, bits [31:0]

Data address for an MemManage fault. This register is updated with the address of a location that produced a MemManage fault. The MMFSR shows the cause of the fault, and whether this field is valid. This field is valid only when MMFSR.MMARVALID is set, otherwise it is UNKNOWN.

In implementations without unique BFAR and MMFAR registers, the value of this register is UNKNOWN if BFSR.BFARVALID is set.

This field resets to an UNKNOWN value on a Warm reset.

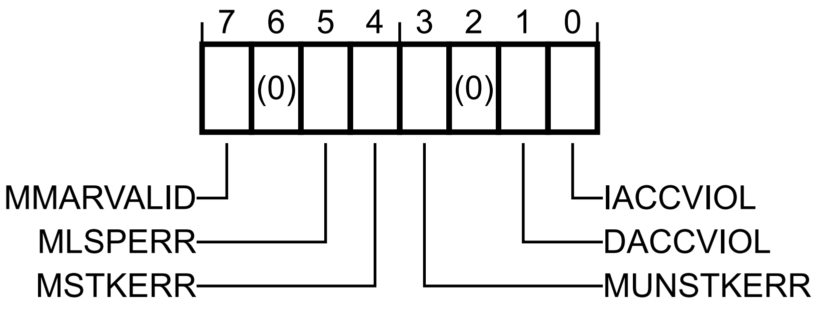
D1.2.146 MMFSR, MemManage Fault Status Register

The MMFSR characteristics are:

Purpose	Shows the status of MPU faults.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. If the Main Extension is implemented, this register is word, halfword, and byte accessible. If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	8-bit read/write-one-to-clear register located at `0xE000ED28`. Secure software can access the Non-secure version of this register via MMFSR_NS located at `0xE002ED28`. The location `0xE002ED28` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states. This register is part of CFSR.

Field descriptions

The MMFSR bit assignments are:



MMARVALID, bit [7]

MMFAR valid flag. Indicates validity of the MMFAR register.

The possible values of this bit are:

0	MMFAR content not valid.
1	MMFAR content valid.

This bit resets to zero on a Warm reset.

Bit [6]

Reserved, RES0.

MLSPERR, bit [5]

MemManage lazy state preservation error flag. Records whether a MemManage fault occurred during FP lazy state preservation.

The possible values of this bit are:

0	No MemManage occurred.
1	MemManage occurred.

This bit resets to zero on a Warm reset.

MSTKERR, bit [4]

MemManage stacking error flag. Records whether a derived MemManage fault occurred during exception entry stacking.

The possible values of this bit are:

0	No derived MemManage occurred.
1	Derived MemManage occurred during exception entry.

This bit resets to zero on a Warm reset.

MUNSTKERR, bit [3]

MemManage unstacking error flag. Records whether a derived MemManage fault occurred during exception return unstacking.

The possible values of this bit are:

0	No derived MemManage fault occurred.
1	Derived MemManage fault occurred during exception return.

This bit resets to zero on a Warm reset.

Bit [2]

Reserved, RES0.

DACCVIOL, bit [1]

Data access violation flag. Records whether a data access violation has occurred.

The possible values of this bit are:

0	No MemManage fault on data access has occurred.
1	MemManage fault on data access has occurred.

A DACCVIOL will be accompanied by an MMFAR update.

This bit resets to zero on a Warm reset.

IACCVIOL, bit [0]

Instruction access violation. Records whether an instruction related memory access violation has occurred.

The possible values of this bit are:

0	No MemManage fault on instruction access has occurred.
1	MemManage fault on instruction access has occurred.

An IACCVIOL is only recorded if a faulted instruction is executed.

This bit resets to zero on a Warm reset.

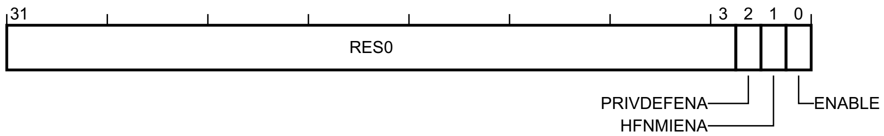
D1.2.147 MPU_CTRL, MPU Control Register

The MPU_CTRL characteristics are:

Purpose	Enables the MPU and, when the MPU is enabled, controls whether the default memory map is enabled as a background region for privileged accesses, and whether the MPU is enabled for HardFaults, NMIs, and exception handlers when FAULTMASK is set to 1.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. Secure software can access the Non-secure version of this register via MPU_CTRL_NS located at `0xE002ED94`. The location `0xE002ED94` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Field descriptions

The MPU_CTRL bit assignments are:



Bits [31:3]

Reserved, RES0.

PRIVDEFENA, bit [2]

Privileged default enable. Controls whether the default memory map is enabled for privileged software.

The possible values of this bit are:

0	Use of default memory map disabled.
1	Use of default memory map enabled for privilege code.

When the ENABLE bit is set to 0, the PE ignores the PRIVDEFENA bit. If no regions are enabled and the PRIVDEFENA and ENABLE bits are set to 1, only privileged code can execute from the system address map. If no MPU regions are implemented this bit is RES0.

This bit resets to zero on a Warm reset.

HFNMIENA, bit [1]

HardFault, NMI enable. Controls whether handlers executing with 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.

D1.2.148 MPU_MAIR0, MPU Memory Attribute Indirection Register 0

The MPU_MAIR0 characteristics are:

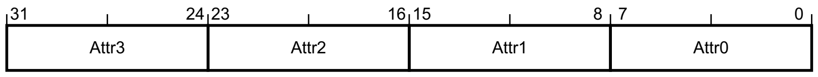
Purpose	Along with MPU_MAIR1, provides the memory attribute encodings corresponding to the AttrIndx values.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000EDC0`. Secure software can access the Non-secure version of this register via MPU_MAIR0_NS located at `0xE002EDC0`. The location `0xE002EDC0` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

Field descriptions

The MPU_MAIR0 bit assignments are:



Attr*m*, bits [8*m*+7:8*m*], for *m* = 0 to 3

Attribute *m*. Memory attribute encoding for MPU regions with an 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.

D1.2.149 MPU_MAIR1, MPU Memory Attribute Indirection Register 1

The MPU_MAIR1 characteristics are:

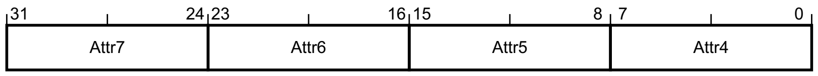
Purpose	Along with MPU_MAIR0, provides the memory attribute encodings corresponding to the AttrIndx values.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000EDC4`. Secure software can access the Non-secure version of this register via MPU_MAIR1_NS located at `0xE002EDC4`. The location `0xE002EDC4` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

This register is RES0 if no MPU regions are implemented in the corresponding Security state.

Field descriptions

The MPU_MAIR1 bit assignments are:



Attr*m*, bits [8(*m*-4)+7:8(*m*-4)], for *m* = 4 to 7

Attribute *m*. Memory attribute encoding for MPU regions with an 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.

D1.2.150 MPU_RBAR, MPU Region Base Address Register

The MPU_RBAR characteristics are:

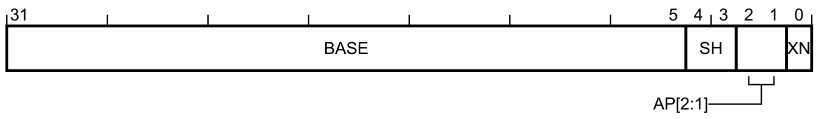
Purpose	Provides indirect read and write access to the base address of the currently selected MPU region for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	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.

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.

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

The MPU_RBAR_A{1..3} characteristics are:

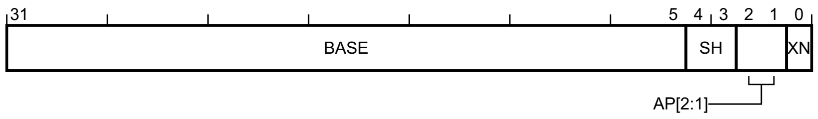
Purpose	Provides indirect read and write access to the base address of the MPU region selected by MPU_RNR[7:2]:(*n*[1:0]) for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. Secure software can access the Non-secure version of this register via MPU_RBAR_An_NS located at `0xE002EDA4` + 8(n-1). The location `0xE002EDA4` + 8(n-1) is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

This register is an alias of the MPU_RBAR register and provides access to the configuration of the MPU region selected by MPU_RNR.REGION had REGION[1:0] been set to *n*[1:0].

Field descriptions

The MPU_RBAR_A{1..3} bit assignments are:



BASE, bits [31:5]

Base address. Contains bits [31:5] of the lower inclusive limit of the selected MPU memory region. This value is zero extended to provide the base address to be checked against.

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.

D1.2.152 MPU_RLAR, MPU Region Limit Address Register

The MPU_RLAR characteristics are:

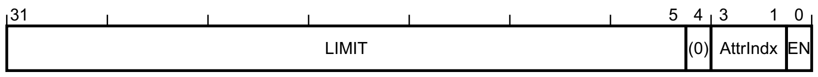
Purpose	Provides indirect read and write access to the limit address of the currently selected MPU region for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. Secure software can access the Non-secure version of this register via MPU_RLAR_NS located at `0xE002EDA0`. The location `0xE002EDA0` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

This register provides access to the configuration of the MPU region selected by MPU_RNR.REGION for the appropriate Security state. The field descriptions apply to the currently selected region.

Field descriptions

The MPU_RLAR bit assignments are:



LIMIT, bits [31:5]

Limit address. Contains bits [31:5] of the upper inclusive limit of the selected MPU memory region. This value is postfixed with `0x1F` to provide the limit address to be checked against.

This field resets to an UNKNOWN value on a Warm reset.

Bit [4]

Reserved, RES0.

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.

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

The MPU_RLAR_A{1..3} characteristics are:

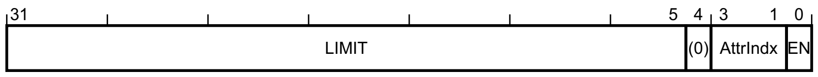
Purpose	Provides indirect read and write access to the limit address of the currently selected MPU region selected by MPU_RNR[7:2]:(*n*[1:0]) for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. 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	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.

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.

Bit [4]

Reserved, RES0.

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.

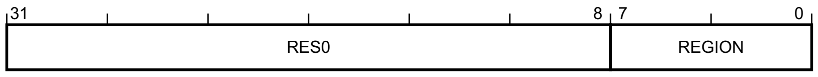
D1.2.154 MPU_RNR, MPU Region Number Register

The MPU_RNR characteristics are:

Purpose	Selects the region currently accessed by MPU_RBAR and MPU_RLAR.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. Secure software can access the Non-secure version of this register via MPU_RNR_NS located at `0xE002ED98`. The location `0xE002ED98` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Field descriptions

The MPU_RNR bit assignments are:



Bits [31:8]

Reserved, RES0.

REGION, bits [7:0]

Region number. Indicates the memory region accessed by MPU_RBAR and MPU_RLAR.

If no MPU regions are implemented, this field is RES0. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE.

This field resets to an UNKNOWN value on a Warm reset.

D1.2.155 MPU_TYPE, MPU Type Register

The MPU_TYPE characteristics are:

Purpose The MPU Type Register indicates how many regions the MPU for the selected Security state supports.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

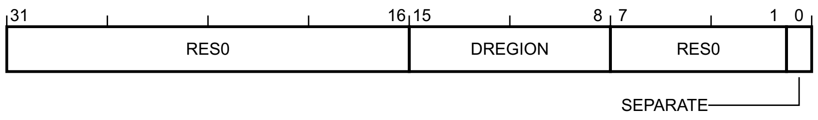
Attributes 32-bit read-only register located at `0xE000ED90`.

Secure software can access the Non-secure version of this register via MPU_TYPE_NS located at `0xE002ED90`. The location `0xE002ED90` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

Field descriptions

The MPU_TYPE bit assignments are:



Bits [31:16]

Reserved, RES0.

DREGION, bits [15:8]

Data regions. Number of regions supported by the MPU.

If this field reads-as-zero, the PE does not implement an MPU for the selected Security state.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [7:1]

Reserved, RES0.

SEPARATE, bit [0]

Separate. Indicates support for separate instructions and data address regions.

Armv8-M only supports unified MPU regions.

This bit reads as zero.

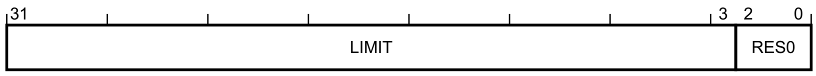
D1.2.156 MSPLIM, Main Stack Pointer Limit Register

The MSPLIM characteristics are:

Purpose	Holds the lower limit of the Main stack pointer.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is banked between Security states.

Field descriptions

The MSPLIM bit assignments are:



LIMIT, bits [31:3]

Stack limit. Bits [31:3] of the Main stack pointer limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure MSPLIM is RAZ/WI.

This field resets to zero on a Warm reset.

Bits [2:0]

Reserved, RES0.

D1.2.157 MVFR0, Media and VFP Feature Register 0

The MVFR0 characteristics are:

Purpose	Describes the features provided by the Floating-point Extension.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Floating-point Extension is implemented. This register is RES0 if the Floating-point Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000EF40`. Secure software can access the Non-secure version of this register via MVFR0_NS located at `0xE002EF40`. The location `0xE002EF40` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Preface

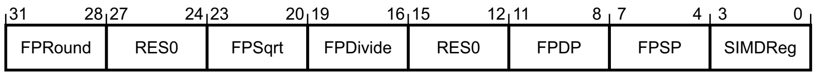
When the Floating-point Extension is not implemented this register reads as `0x00000000`.

Where single-precision only floating-point is supported this register reads as `0x10110021`.

Where single and double-precision floating-point are supported this register reads as `0x10110221`.

Field descriptions

The MVFR0 bit assignments are:



FPRound, bits [31:28]

Floating-point rounding modes. Indicates the rounding modes supported by the Floating-point Extension.

The possible values of this field are:

`0b0001` All rounding modes supported.

All other values are reserved.

This field reads as `0b0001`.

Bits [27:24]

Reserved, RES0.

FPSqrt, bits [23:20]

Floating-point square root. Indicates the support for floating-point square root operations.

The possible values of this field are:

`0b0001` Supported.

All other values are reserved.

This field reads as `0b0001`.

FPDivide, bits [19:16]

Floating-point divide. Indicates the support for floating-point divide operations.

The possible values of this field are:

`0b0001` Supported.

All other values are reserved.

This field reads as `0b0001`.

Bits [15:12]

Reserved, RES0.

FPDP, bits [11:8]

Floating-point double-precision. Indicates support for floating-point double-precision operations.

The possible values of this field are:

`0b0000`	Not supported.
`0b0010`	Supported.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

FPSP, bits [7:4]

Floating-point single-precision. Indicates support for floating-point single-precision operations.

The possible values of this field are:

`0b0010`	Supported.

All other values are reserved.

This field reads as `0b0010`.

SIMDReg, bits [3:0]

SIMD registers. Indicates size of Floating-Point Extension register file.

The possible values of this field are:

`0b0001`	16 x 64-bit registers.

All other values are reserved.

This field reads as `0b0001`.

D1.2.158 MVFR1, Media and VFP Feature Register 1

The MVFR1 characteristics are:

Purpose	Describes the features provided by the Floating-point Extension.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Floating-point Extension is implemented. This register is RES0 if the Floating-point Extension is not implemented.
Attributes	32-bit read-only register located at `0xE000EF44`. Secure software can access the Non-secure version of this register via MVFR1_NS located at `0xE002EF44`. The location `0xE002EF44` is RES0 to software executing in Non-secure state and the debugger. This register is not banked between Security states.

Preface

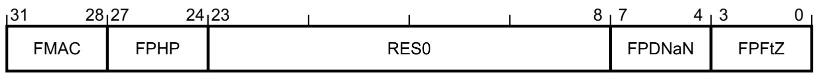
When floating-point is not implemented this register reads as `0x00000000`.

Where single-precision only floating-point is supported this register reads as `0x11000011`.

Where single and double-precision floating-point are supported this register reads as `0x12000011`.

Field descriptions

The MVFR1 bit assignments are:



FMAC, bits [31:28]

Fused multiply accumulate. Indicates whether the Floaing-point Extension implements the fused multiply accumulate instructions.

The possible values of this field are:

`0b0001`	Implemented.

All other values are reserved.

This field reads as `0b0001`.

FPHP, bits [27:24]

Floating-point half-precision. Indicates whether the Floating-point Extension implements half-precision floating-point conversion instructions.

The possible values of this field are:

`0b0001`	Half-precision to single-precision implemented.
`0b0010`	Half-precision to single and double-precision implemented.

All other values are reserved.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [23:8]

Reserved, RES0.

FPDNaN, bits [7:4]

Floating-point default NaN. Indicates whether the Floating-point Extension implementation supports NaN propagation.

The possible values of this field are:

`0b0001`	Propagation of NaN values supported.

All other values are reserved.

This field reads as `0b0001`.

FPFtZ, bits [3:0]

Floating-point flush-to-zero. Indicates whether subnormals are always flushed-to-zero.

The possible values of this field are:

`0b0001`	Full denormalized numbers arithmetic supported.

All other values are reserved.

This field reads as `0b0001`.

D1.2.159 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.

Configurations Present only if the Floating-point Extension is implemented.

This register is RES0 if the Floating-point Extension is not implemented.

Attributes 32-bit read-only register located at `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.

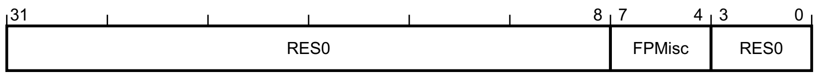
Preface

When floating-point is not implemented this register reads as `0x00000000`.

When floating-point is implemented this register reads as `0x00000040`.

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:

`0b0100` Selection, directed conversion to integer, `VMINNM` and `VMAXNM` supported.

All other values are reserved.

This field reads as `0b0100`.

Bits [3:0]

Reserved, RES0.

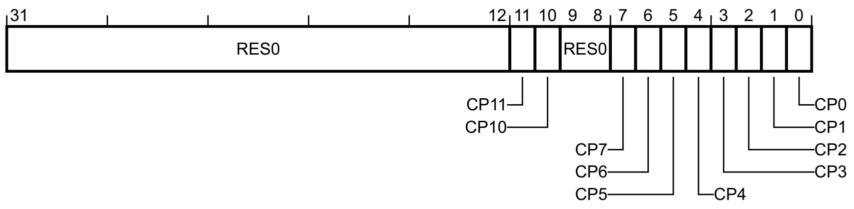
D1.2.160 NSACR, Non-secure Access Control Register

The NSACR characteristics are:

Purpose	Defines the Non-secure access permissions for both the FP Extension and coprocessors CP0 to CP7.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write register located at `0xE000ED8C`. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The NSACR bit assignments are:



Bits [31:12]

Reserved, RES0.

CP11, bit [11]

CP11 access. Enables Non-secure access to the Floating-point Extension.

Programming with a different value than that used for CP10 is UNPREDICTABLE. If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

CP10, bit [10]

CP10 access. Enables Non-secure access to the Floating-point Extension.

The possible values of this bit are:

0	Non-secure accesses to the Floating-point Extension generate a NOCP UsageFault.
1	Non-secure access to the Floating-point Extension permitted.

If the Floating-point Extension is not implemented, this bit is RAZ/WI.

This bit resets to an UNKNOWN value on a Warm reset.

Bits [9:8]

Reserved, RES0.

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.

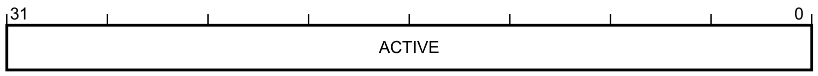
D1.2.161 NVIC_IABRn, Interrupt Active Bit Register, n = 0 - 15

The NVIC_IABR{0..15} characteristics are:

Purpose	For each group of 32 interrupts, shows the active state of each interrupt.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000E300` + 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.

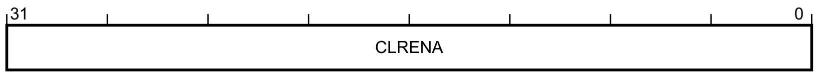
D1.2.162 NVIC_ICERn, Interrupt Clear Enable Register, n = 0 - 15

The NVIC_ICER{0..15} characteristics are:

Purpose	Clears or reads the enabled state of each group of 32 interrupts.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write-one-to-clear register located at `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.

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

The NVIC_ICPR{0..15} characteristics are:

Purpose Clears or reads the pending state of each group of 32 interrupts.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

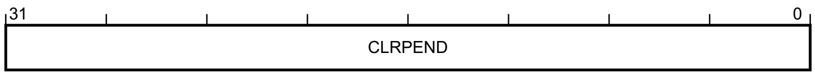
Attributes 32-bit read/write-one-to-clear register located at `0xE000E280` + 4*n*.

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.

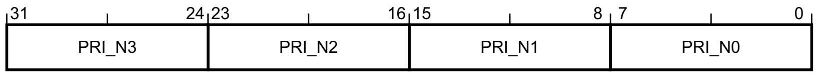
D1.2.164 NVIC_IPRn, Interrupt Priority Register, n = 0 - 123

The NVIC_IPR{0..123} characteristics are:

Purpose	Sets or reads interrupt priorities.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. If the Main Extension is implemented, this register is word, halfword, and byte accessible. If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000E400` + 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.

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*+3 targets Secure state, this field is RAZ/WI from Non-secure.

This field resets to zero on a Warm reset.

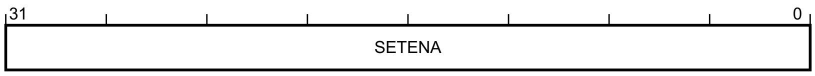
D1.2.165 NVIC_ISERn, Interrupt Set Enable Register, n = 0 - 15

The NVIC_ISER{0..15} characteristics are:

Purpose	Enables or reads the enabled state of each group of 32 interrupts.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write-one-to-set register located at `0xE000E100` + 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.

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

The NVIC_ISPR{0..15} characteristics are:

Purpose Enables or reads the pending state of each group of 32 interrupts.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

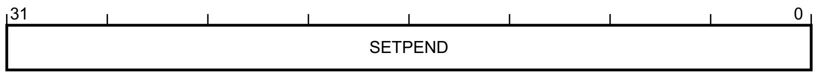
Attributes 32-bit read/write-one-to-set register located at `0xE000E200` + 4*n*.

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.

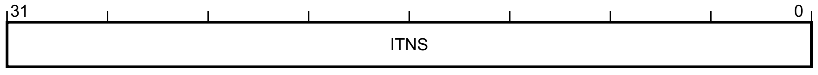
D1.2.167 NVIC_ITNSn, Interrupt Target Non-secure Register, n = 0 - 15

The NVIC_ITNS{0..15} characteristics are:

Purpose	For each group of 32 interrupts, determines whether each interrupt targets Non-secure or Secure state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000E380` + 4*n*. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The NVIC_ITNS{0..15} bit assignments are:



ITNS, bits [31:0]

Interrupt Targets Non-secure. For ITNS[*m*] in NVIC_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.

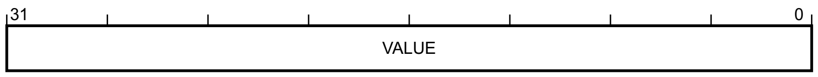
D1.2.168 PC, Program Counter

The PC characteristics are:

Purpose	Holds the current Program Counter value.
Usage constraints	Privileged and unprivileged access permitted.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The PC bit assignments are:



VALUE, bits [31:0]

Program Counter. Holds the address of the current instruction.

Software can refer to PC as R15.

This field resets to an UNKNOWN value on a Warm reset.

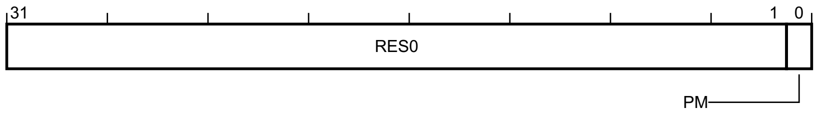
D1.2.169 PRIMASK, Exception Mask Register

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.

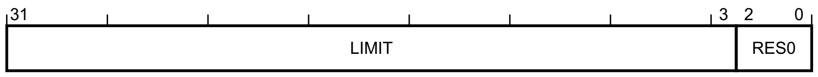
D1.2.170 PSPLIM, Process Stack Pointer Limit Register

The PSPLIM characteristics are:

Purpose	Holds the lower limit for the Process stack pointer.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is banked between Security states.

Field descriptions

The PSPLIM bit assignments are:



LIMIT, bits [31:3]

Stack limit. Bits [31:3] of the Process stack limit address for the selected Security state.

Many instructions and exception entry will generate an exception if the appropriate stack pointer would be updated to a value lower than this limit. If the Main Extension is not implemented, the Non-secure PSPLIM is RAZ/WI.

This field resets to zero on a Warm reset.

Bits [2:0]

Reserved, RES0.

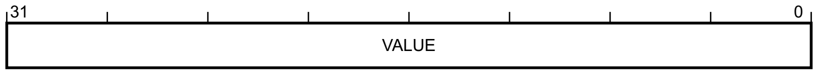
D1.2.171 Rn, General-Purpose Register, n = 0 - 12

The R{0..12} characteristics are:

Purpose	General-purpose register.
Usage constraints	Both privileged and unprivileged accesses are permitted.
Configurations	This register is always implemented.
Attributes	32-bit read/write register. This register is not banked between Security states.

Field descriptions

The R{0..12} bit assignments are:



VALUE, bits [31:0]

General purpose register value. Armv8-M implemented thirteen general-purpose 32-bit registers, R0 to R12.

This field resets to an UNKNOWN value on a Warm reset.

D1.2.172 RETPSR, Combined Exception Return Program Status Registers

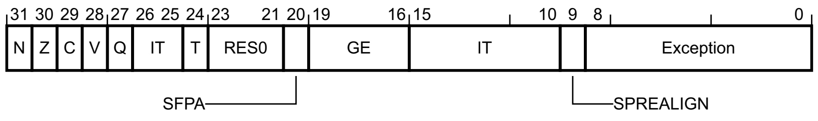
The RETPSR characteristics are:

Purpose	Value pushed to the stack on exception entry. On exception return this is used to restore the flags and other architectural state. This payload is also used for FNC_RETURN stacking, however in this case only some of the fields are used. See FunctionReturn() for details.
Usage constraints	None.
Configurations	All.
Attributes	32-bit payload.

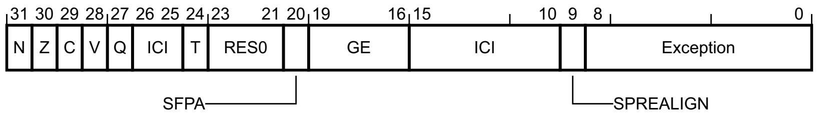
Field descriptions

The RETPSR bit assignments are:

When `{RETPSR[26:25], RETPSR[11:10]} != 0`:



When `{RETPSR[26:25], RETPSR[11:10]} == 0`:



N, bit [31]

Negative flag. Value corresponding to APSR.N.

Z, bit [30]

Zero flag. Value corresponding to APSR.Z.

C, bit [29]

Carry flag. Value corresponding to APSR.C.

V, bit [28]

Overflow flag. Value corresponding to APSR.V.

Q, bit [27]

Saturate flag. Value corresponding to APSR.Q.

T, bit [24]

T32 state. Value corresponding to EPSR.T.

Bits [23:21]

Reserved, RES0.

SFPA, bit [20]

Secure floating-point active. Value corresponding to CONTROL.SFPA.

GE, bits [19:16]

Greater-than or equal flag. Value corresponding to APSR.GE.

IT, bits [15:10,26:25], when {RETPSR[26:25], RETPSR[11:10]} != 0

If-then flags. Value corresponding to EPSR.IT.

ICI, bits [26:25,15:10], when {RETPSR[26:25], RETPSR[11:10]} == 0

Interrupt continuation flags. Value corresponding to EPSR.ICI.

SPREALIGN, bit [9]

Stack-pointer re-align. Indicates whether the SP was re-aligned to an 8-byte alignment on exception entry.

The possible values of this bit are:

0	The stack pointer was 8-byte aligned before exception entry began, no special handling is required on exception return.
1	The stack pointer was only 4-byte aligned before exception entry. The exception entry realigned SP to 8-byte alignment by increasing the stack frame size by 4-bytes.

Exception, bits [8:0]

Exception number. Value corresponding to IPSR.Exception.

D1.2.173 SAU_CTRL, SAU Control Register

The SAU_CTRL characteristics are:

Purpose	Allows enabling of the Security Attribution Unit.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

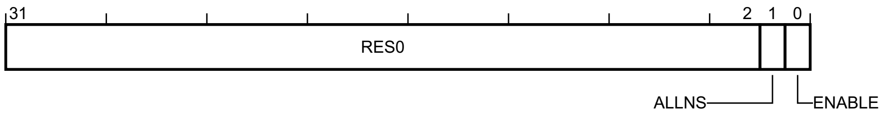
Preface

It is IMPLEMENTATION DEFINED whether this register:

- Resets to `0x0` - in this case SAU_REGIONn registers are UNKNOWN at reset.
- Resets to an IMPLEMENTATION DEFINED value.

Field descriptions

The SAU_CTRL bit assignments are:



Bits [31:2]

Reserved, RES0.

ALLNS, bit [1]

All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure.

The possible values of this bit are:

0	Memory is marked as Secure and is not Non-secure callable.
1	Memory is marked as Non-secure.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

ENABLE, bit [0]

Enable. Enables the SAU.

The possible values of this bit are:

0	The SAU is disabled.
1	The SAU is enabled.

If this register resets to 1, the SAU region registers also reset to an IMPLEMENTATION DEFINED value.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

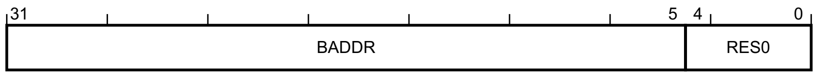
D1.2.174 SAU_RBAR, SAU Region Base Address Register

The SAU_RBAR characteristics are:

Purpose	Provides indirect read and write access to the base address of the currently selected SAU region.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SAU_RBAR bit assignments are:



BADDR, bits [31:5]

Base address. Holds bits [31:5] of the base address for the selected SAU region.

Bits [4:0] of the base address are defined as `0x00`.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

Bits [4:0]

Reserved, RES0.

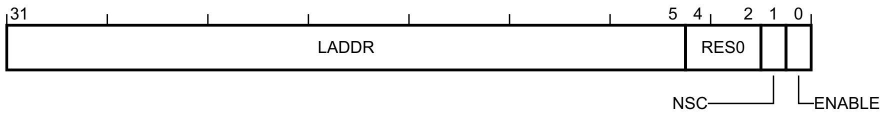
D1.2.175 SAU_RLAR, SAU Region Limit Address Register

The SAU_RLAR characteristics are:

Purpose	Provides indirect read and write access to the limit address of the currently selected SAU region.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SAU_RLAR bit assignments are:



LADDR, bits [31:5]

Limit address. Holds bits [31:5] of the limit address for the selected SAU region.

Bits [4:0] of the limit address are defined as `0x1F`.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

Bits [4:2]

Reserved, RES0.

NSC, bit [1]

Non-secure callable. Controls whether Non-secure state is permitted to execute an SG instruction from this region.

The possible values of this bit are:

0	Region is not Non-secure callable.
1	Region is Non-secure callable.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

ENABLE, bit [0]

Enable. SAU region enable.

The possible values of this bit are:

0	SAU region is disabled.
1	SAU region is enabled.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

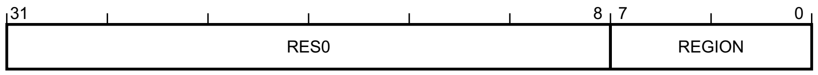
D1.2.176 SAU_RNR, SAU Region Number Register

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	To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SAU_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.

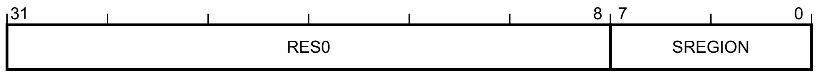
D1.2.177 SAU_TYPE, SAU Type Register

The SAU_TYPE characteristics are:

Purpose	Indicates the number of regions implemented by the Security Attribution Unit.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read-only register located at `0xE000EDD4`. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SAU_TYPE bit assignments are:



Bits [31:8]

Reserved, RES0.

SREGION, bits [7:0]

SAU regions. The number of implemented SAU regions.

If this field is RAZ, the SAU behaves as follows:

- SAU_CTRL.ENABLE behaves as RAZ/WI.
- It is IMPLEMENTATION DEFINED whether SAU_CTRL.ALLNS behaves as RAO/WI and all attribution is performed by the IDAU.
- SAU_RNR, SAU_RBAR, and SAU_RLAR behave as RAZ/WI.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.178 SCR, System Control Register

The SCR characteristics are:

Purpose Sets or returns system control data.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

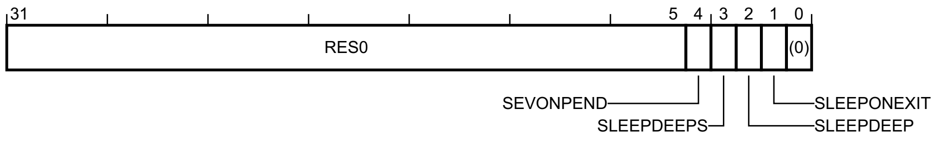
Attributes 32-bit read/write register located at `0xE000ED10`.

Secure software can access the Non-secure version of this register via SCR_NS located at `0xE002ED10`. The location `0xE002ED10` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

Field descriptions

The SCR bit assignments are:



Bits [31:5]

Reserved, RES0.

SEVONPEND, bit [4]

Send event on pend. Determines whether an interrupt assigned to the same Security state as the SEVONPEND bit transitioning from inactive state to pending state generates a wakeup event.

This bit is banked between Security states.

The possible values of this bit are:

0	Transitions from inactive to pending are not wakeup events.
1	Transitions from inactive to pending are wakeup events.

This bit resets to zero on a Warm reset.

SLEEPDEEPS, bit [3]

Sleep deep secure. This field controls whether the SLEEPDEEP bit is only accessible from the Secure state.

This bit is not banked between Security states.

The possible values of this bit are:

0	The SLEEPDEEP bit accessible from both Security states.
1	The SLEEPDEEP bit behaves as RAZ/WI when accessed from the Non-secure state.

This bit is only accessible from the Secure state, and behaves as RAZ/WI when accessed from the Non-secure state. If a PE does not implement the deep sleep state this bit behaves as RAZ/WI from both Security states.

This bit is RAZ/WI from Non-secure state.

This bit resets to zero on a Warm reset.

SLEEPDEEP, bit [2]

Sleep deep. Provides a qualifying hint indicating that waking from sleep might take longer. An implementation can use this bit to select between two alternative sleep states.

This bit is not banked between Security states.

The possible values of this bit are:

0	Selected sleep state is not deep sleep.
1	Selected sleep state is deep sleep.

Details of the implemented sleep states, if any, and details of the use of this bit, are IMPLEMENTATION DEFINED. If the PE does not implement a deep sleep state then this bit can be RAZ/WI.

This bit resets to zero on a Warm reset.

SLEEPONEXIT, bit [1]

Sleep on exit. Determines whether, on an exit from an ISR that returns to the base level of execution priority, the PE enters a sleep state.

This bit is banked between Security states.

The possible values of this bit are:

0	Enter sleep state disabled.
1	Enter sleep state permitted.

The Secure version of this field is used if the Background state being returned to is the Secure state, otherwise the Non-secure version is used.

This bit resets to zero on a Warm reset.

Bit [0]

Reserved, RES0.

D1.2.179 SFAR, Secure Fault Address Register

The SFAR characteristics are:

Purpose	Shows the address of the memory location that caused a Security violation.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write register located at `0xE000EDE8`. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SFAR bit assignments are:

31
0
ADDRESS

ADDRESS, bits [31:0]

Address. The address of an access that caused an attribution unit violation. This field is only valid when SFSR.SFARVALID is set. This allows the actual flip flops associated with this register to be shared with other fault address registers. If an implementation chooses to share the storage in this way, care must be taken to not leak Secure address information to the Non-secure state. One way of achieving this is to share the SFAR register with the MMFAR_S register, which is not accessible to the Non-secure state.

This field resets to an UNKNOWN value on a Warm reset.

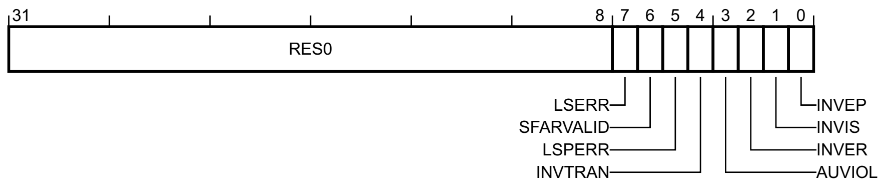
D1.2.180 SFSR, Secure Fault Status Register

The SFSR characteristics are:

Purpose	Provides information about any security related faults.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	32-bit read/write-one-to-clear register located at `0xE000EDE4`. This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

Field descriptions

The SFSR bit assignments are:



Bits [31:8]

Reserved, RES0.

LSERR, bit [7]

Lazy state error flag. Sticky flag indicating that an error occurred during lazy state activation or deactivation.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

SFARVALID, bit [6]

Secure fault address valid. This bit is set when the SFAR register contains a valid value. As with similar fields, such as BFSR.BFARVALID and MMFSR.MMARVALID, this bit can be cleared by other exceptions, such as BusFault.

The possible values of this bit are:

0	SFAR content not valid.
1	SFAR content valid.

This bit resets to zero on a Warm reset.

LSPERR, bit [5]

Lazy state preservation error flag. 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.

D1.2.181 SHCSR, System Handler Control and State Register

The SHCSR characteristics are:

Purpose	Provides access to the active and pending status of system exceptions.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	This register is always implemented.
Attributes	32-bit read/write register located at `0xE000ED24`. Secure software can access the Non-secure version of this register via SHCSR_NS located at `0xE002ED24`. The location `0xE002ED24` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states on a bit by bit basis.

Preface

Exception processing automatically updates the SHCSR fields. However, software can write to the register to add or remove the pending or active state of an exception. When updating the SHCSR, Arm recommends using a read-modify-write sequence, to avoid unintended effects on the state of the exception handlers.

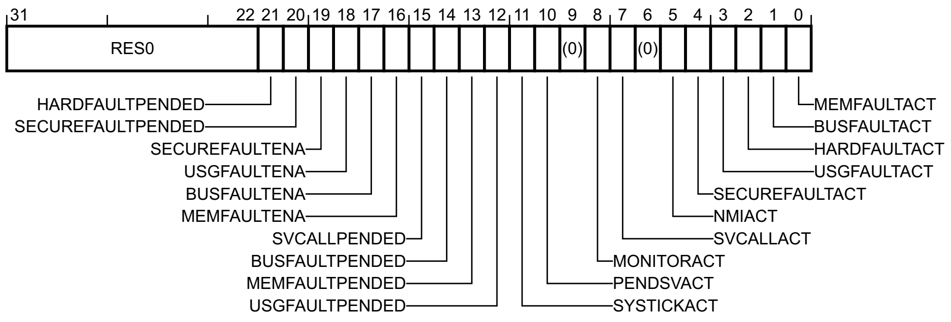
Removing the active state of an exception can change the current execution priority, and affect the exception return consistency checks. If software removes the active state, causing a change in current execution priority, this can defeat the architectural behavior that prevents an exception from preempting its own handler.

Pending state bits are set to one when an exception occurs and are cleared to zero when the exception becomes active.

Active state bits are set to one when the associated exception becomes active.

Field descriptions

The SHCSR bit assignments are:



Bits [31:22]

Reserved, RES0.

HARDFAULTPENDED, bit [21]

HardFault exception pended state. This bit indicates and allows modification of the pending state of the HardFault exception corresponding to the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

0	HardFault exception not pending for the selected Security state.
1	HardFault exception pending for the selected Security state.

The Non-secure view of this bit is RAZ/WI if AIRCR.BFHFNMINS is zero.

This bit resets to zero on a Warm reset.

Note

The Non-secure HardFault exception will not preempt if AIRCR.BFHFNMINS is set to zero.

SECUREFAULTPENDED, bit [20]

SecureFault exception pended state. This bit indicates and allows modification of the pending state of the SecureFault exception.

This bit is not banked between Security states.

The possible values of this bit are:

0	SecureFault exception not pending.
1	SecureFault exception pending.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

SECUREFAULTENA, bit [19]

SecureFault exception enable. The value of this bit defines whether the SecureFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

0	SecureFault exception disabled.
1	SecureFault exception enabled.

When disabled, exceptions that target SecureFault escalate to Secure state HardFault.

This bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

USGFAULTENA, bit [18]

UsageFault exception enable. The value of this bit defines whether the UsageFault exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

0	UsageFault exception disabled for the selected Security state.
1	UsageFault exception enabled for the selected Security state.

When the UsageFault exception is disabled, exceptions targeting UsageFault escalate to HardFault.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

BUSFAULTENA, bit [17]

BusFault exception enable. The value of this bit defines whether the BusFault exception is enabled.

This bit is not banked between Security states.

The possible values of this bit are:

0	BusFault exception disabled.
1	BusFault exception enabled.

The BusFault exception is not banked between Security states. When the BusFault exception is disabled, exceptions targeting BusFault escalate to HardFault.

If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

MEMFAULTENA, bit [16]

MemManage exception enable. The value of this bit defines whether the MemManage exception is enabled for the selected Security state.

This bit is banked between Security states.

The possible values of this bit are:

0	MemManage exception disabled for the selected Security state.
1	MemManage exception enabled for the selected Security state.

If the Main Extension is not implemented, this bit is RES0.

This bit resets to zero on a Warm reset.

Note

When the MemManage exception is disabled, exceptions targeting MemManage escalate to 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 AIRCR.STTNS is zero, or the Non-secure state3 if AIRCR.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.

D1.2.182 SHPR1, System Handler Priority Register 1

The SHPR1 characteristics are:

Purpose Sets or returns priority for system handlers 4 - 7.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if the Main Extension is implemented.

This register is RES0 if the Main Extension is not implemented.

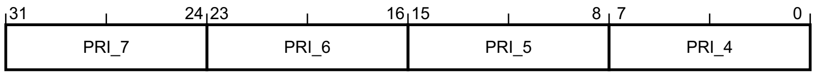
Attributes 32-bit read/write register located at `0xE000ED18`.

Secure software can access the Non-secure version of this register via SHPR1_NS located at `0xE002ED18`. The location `0xE002ED18` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

Field descriptions

The SHPR1 bit assignments are:



PRI_7, bits [31:24]

Priority 7. Priority of system handler 7, SecureFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

PRI_6, bits [23:16]

Priority 6. Priority of system handler 6, UsageFault.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

PRI_5, bits [15:8]

Priority 5. Priority of system handler 5, BusFault.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

This field resets to zero on a Warm reset.

PRI_4, bits [7:0]

Priority 4. Priority of system handler 4, MemManage.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

D1.2.183 SHPR2, System Handler Priority Register 2

The SHPR2 characteristics are:

Purpose Sets or returns priority for system handlers 8 - 11.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

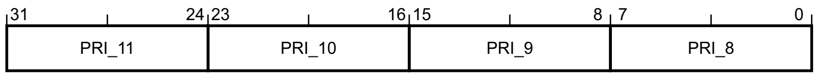
Attributes 32-bit read/write register located at `0xE000ED1C`.

Secure software can access the Non-secure version of this register via SHPR2_NS located at `0xE002ED1C`. The location `0xE002ED1C` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

Field descriptions

The SHPR2 bit assignments are:



PRI_11, bits [31:24]

Priority 11. Priority of system handler 11, 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.

D1.2.184 SHPR3, System Handler Priority Register 3

The SHPR3 characteristics are:

Purpose Sets or returns priority for system handlers 12 - 15.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

If the Main Extension is implemented, this register is word, halfword, and byte accessible.

If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

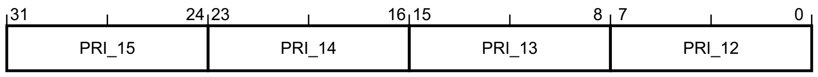
Attributes 32-bit read/write register located at `0xE000ED20`.

Secure software can access the Non-secure version of this register via SHPR3_NS located at `0xE002ED20`. The location `0xE002ED20` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states on a bit by bit basis.

Field descriptions

The SHPR3 bit assignments are:



PRI_15, bits [31:24]

Priority 15. Priority of system handler 15, SysTick.

If two SysTick timers are implemented this field is banked between Security states.

If less than two SysTick timers are implemented this field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0. If one timer is implemented, this field corresponds to the Secure state if AIRCR.STTNS is zero, or the Non-secure state if AIRCR.STTNS is one.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to zero on a Warm reset.

PRI_14, bits [23:16]

Priority 14. Priority of system handler 14, PendSV.

This field is banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

This field resets to zero on a Warm reset.

PRI_13, bits [15:8]

Reserved, RES0.

PRI_12, bits [7:0]

Priority 12. Priority of system handler 12, DebugMonitor.

This field is not banked between Security states.

If the PE implements fewer than 8 bits of priority, then the least significant bits of this field are RES0.

If DEMCR.SDME is one this field is RAZ/WI from Non-secure state

If the Main Extension is not implemented, this field is RES0.

This field resets to zero on a Warm reset.

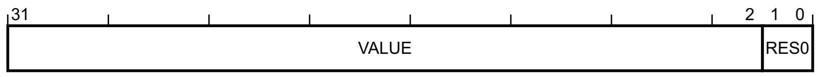
D1.2.185 SP, Current Stack Pointer Register

The SP characteristics are:

Purpose	Exception and procedure stack pointer register.
Usage constraints	Privileged and unprivileged access permitted.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The SP bit assignments are:



VALUE, bits [31:2]

Stack pointer. Holds bits[31:2] of the stack pointer address. The current stack pointer is selected from one of MSP_NS, PSP_NS, MSP_S or PSP_S.

Software can refer to SP as R13.

This field resets to an UNKNOWN value on a Warm reset.

Bits [1:0]

Reserved, RES0.

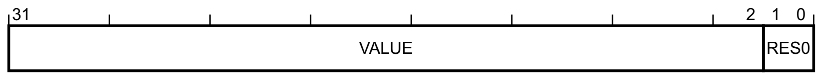
D1.2.186 SP_NS, Stack Pointer (Non-secure)

The SP_NS characteristics are:

Purpose	Provides access to the current Non-secure stack pointer.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

Field descriptions

The SP_NS bit assignments are:



VALUE, bits [31:2]

Stack pointer. Holds bits[31:2] of the current Non-secure stack pointer address. SP_NS is selected from one of MSP_NS or PSP_NS. Access to SP_NS is provided via MRS and MSR and is subject to stack limit checking.

This field resets to an UNKNOWN value on a Warm reset.

Bits [1:0]

Reserved, RES0.

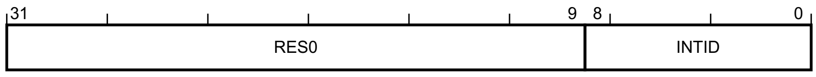
D1.2.187 STIR, Software Triggered Interrupt Register

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.

D1.2.188 SYST_CALIB, SysTick Calibration Value Register

The SYST_CALIB characteristics are:

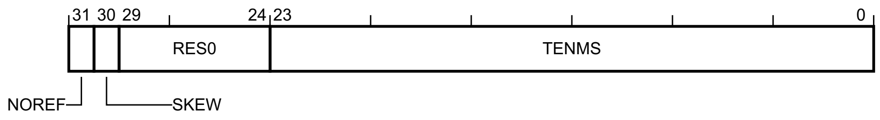
Purpose	Reads the SysTick timer calibration value and parameters for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if at least one SysTick timer is implemented. This register is RES0 if no SysTick timer is implemented.
Attributes	32-bit read-only register located at `0xE000E01C`. Secure software can access the Non-secure version of this register via SYST_CALIB_NS located at `0xE002E01C`. The location `0xE002E01C` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

If the Main Extension is implemented then, two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

Field descriptions

The SYST_CALIB bit assignments are:



NOREF, bit [31]

No reference. Indicates whether the IMPLEMENTATION DEFINED reference clock is implemented.

The possible values of this bit are:

0	Reference clock is implemented.
1	Reference clock is not implemented.

When this bit is 1, the CLKSOURCE bit of the SYST_CSR register is forced to 1 and cannot be cleared to 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This bit reads as an IMPLEMENTATION DEFINED value.

SKEW, bit [30]

Skew. Indicates whether the 10ms calibration value is exact.

The possible values of this bit are:

0	TENMS calibration value is exact.
1	TENMS calibration value is inexact.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This bit reads as an IMPLEMENTATION DEFINED value.

Bits [29:24]

Reserved, RES0.

TENMS, bits [23:0]

Ten milliseconds. Optionally holds a reload value to be used for 10ms (100Hz) timing, subject to system clock skew errors. If this field is zero, the calibration value is not known.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.189 SYST_CSR, SysTick Control and Status Register

The SYST_CSR characteristics are:

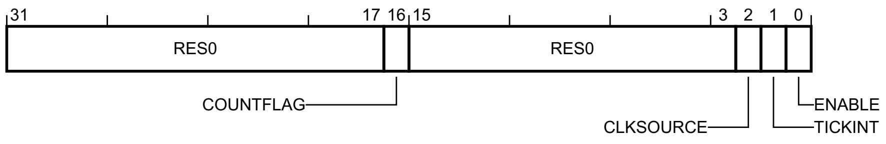
Purpose	Controls the SysTick timer and provides status data for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if at least one SysTick timer is implemented. This register is RES0 if no SysTick timer is implemented.
Attributes	32-bit read/write register located at `0xE000E010`. Secure software can access the Non-secure version of this register via SYST_CSR_NS located at `0xE002E010`. The location `0xE002E010` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

Field descriptions

The SYST_CSR bit assignments are:



Bits [31:17]

Reserved, RES0.

COUNTFLAG, bit [16]

Count flag. Indicates whether the counter has counted to zero since the last read of this register.

The possible values of this bit are:

0	Timer has not counted to 0.
1	Timer has counted to 0.

COUNTFLAG is set to 1 by a count transition from 1 to 0. COUNTFLAG is cleared to 0 if software reads this bit as one, and by any write to the SYST_CVR for the selected Security state. Debugger reads do not clear the COUNTFLAG.

If set this bit clears to zero when read by software. Reads from the debugger do not clear this bit.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

Bits [15:3]

Reserved, RES0.

CLKSOURCE, bit [2]

Clock source. Indicates the SysTick clock source.

The possible values of this bit are:

0	Uses the IMPLEMENTATION DEFINED external reference clock.
1	Uses the PE clock.

If no external clock is implemented, this bit reads as 1 and ignores writes.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to an IMPLEMENTATION DEFINED value on a Warm reset.

TICKINT, bit [1]

Tick interrupt. Indicates whether counting to 0 causes the status of the SysTick exception to change to pending.

The possible values of this bit are:

0	Count to 0 does not affect the SysTick exception status.
1	Count to 0 changes the SysTick exception status to pending.

Changing the value of the counter to 0 by writing the SysTick does not change the status of the SysTick exception.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

ENABLE, bit [0]

SysTick enable. Indicates the enabled status of the SysTick counter.

The possible values of this bit are:

0	Counter is disabled.
1	Counter is enabled.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this bit is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this bit is RES0.

This bit resets to zero on a Warm reset.

D1.2.190 SYST_CVR, SysTick Current Value Register

The SYST_CVR characteristics are:

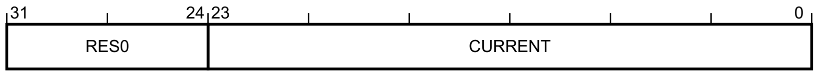
Purpose	Reads or clears the SysTick timer current counter value for the selected Security state.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if at least one SysTick timer is implemented. This register is RES0 if no SysTick timer is implemented.
Attributes	32-bit read/write-to-clear register located at `0xE000E018`. Secure software can access the Non-secure version of this register via SYST_CVR_NS located at `0xE002E018`. The location `0xE002E018` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states.

Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both aliases of this register behave as RES0.

Field descriptions

The SYST_CVR bit assignments are:



Bits [31:24]

Reserved, RES0.

CURRENT, bits [23:0], on a read

Current counter value. Provides the value of the SysTick timer counter for the selected Security state.

It is IMPLEMENTATION DEFINED whether the current counter value decrements if the PE is sleeping and SCR.SLEEPDEEP is set.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

CURRENT, bits [23:0], on a write

Reset counter value. Writing any value clears the SysTick timer counter for the selected Security state to zero.

D1.2.191 SYST_RVR, SysTick Reload Value Register

The SYST_RVR characteristics are:

Purpose Provides access SysTick timer counter reload value for the selected Security state.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations Present only if at least one SysTick timer is implemented.

This register is RES0 if no SysTick timer is implemented.

Attributes 32-bit read/write register located at `0xE000E014`.

Secure software can access the Non-secure version of this register via SYST_RVR_NS located at `0xE002E014`. The location `0xE002E014` is RES0 to software executing in Non-secure state and the debugger.

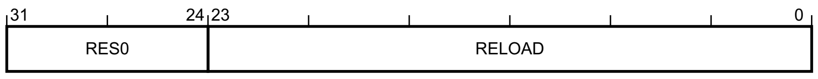
This register is banked between Security states.

Preface

If the Main Extension is implemented, then two SysTick timers are implemented. If the Main Extension is not implemented, then it is IMPLEMENTATION DEFINED whether none, one or two SysTick timers are implemented. Where two SysTick timers are implemented, this register is banked. Where one SysTick timer is implemented, this register is not banked, and Non-secure accesses behave as RAZ/WI if ICSR.STTNS is clear. If no SysTick timer is implemented, both instances of this register behave as RES0.

Field descriptions

The SYST_RVR bit assignments are:



Bits [31:24]

Reserved, RES0.

RELOAD, bits [23:0]

Counter reload value. The value to load into the SYST_CVR for the selected Security state when the counter reaches 0.

If only one SysTick timer is implemented and ICSR.STTNS is clear, this field is RAZ/WI from Non-secure state.

If no SysTick timer is implemented this field is RES0.

This field resets to an UNKNOWN value on a Warm reset.

D1.2.192 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.

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.

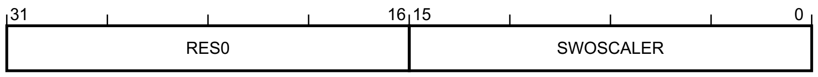
This register is RES0 if the TPIU is not implemented or does not support SWO.

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.

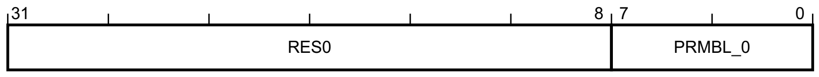
D1.2.193 TPIU_CIDR0, TPIU Component Identification Register 0

The TPIU_CIDR0 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FF0`. This register is not banked between Security states.

Field descriptions

The TPIU_CIDR0 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_0, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0D`.

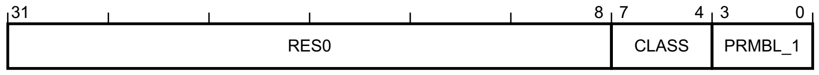
D1.2.194 TPIU_CIDR1, TPIU Component Identification Register 1

The TPIU_CIDR1 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FF4`. This register is not banked between Security states.

Field descriptions

The TPIU_CIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

CLASS, bits [7:4]

CoreSight component class. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x9`.

PRMBL_1, bits [3:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x0`.

D1.2.195 TPIU_CIDR2, TPIU Component Identification Register 2

The TPIU_CIDR2 characteristics are:

Purpose Provides CoreSight discovery information for the TPIU.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

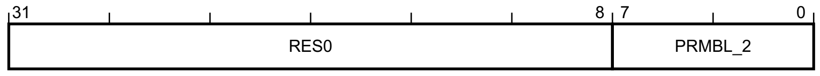
This register is RES0 if the TPIU is not implemented.

Attributes 32-bit read-only register located at `0xE0040FF8`.

This register is not banked between Security states.

Field descriptions

The TPIU_CIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

PRMBL_2, bits [7:0]

CoreSight component identification preamble. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as `0x05`.

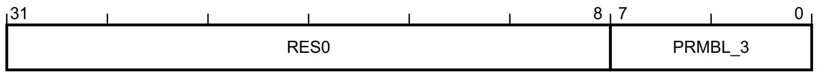
D1.2.196 TPIU_CIDR3, TPIU Component Identification Register 3

The TPIU_CIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `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`.

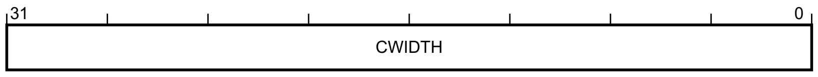
D1.2.197 TPIU_CSPSR, TPIU Current Parallel Port Sizes Register

The TPIU_CSPSR characteristics are:

Purpose	Controls the width of the parallel trace port.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read/write register located at `0xE0040004`. This register is not banked between Security states.

Field descriptions

The TPIU_CSPSR bit assignments are:



CWIDTH, bits [31:0]

Current width. CWIDTH[*m*] represents a parallel trace port width of (*m*+1).

The possible values of each bit are:

0	Width (N+1) is not the current parallel trace port width.
1	Width (N+1) is the current parallel trace port width.

A debugger must set only one bit is set to 1, and all others must be zero. The effect of writing a value with more than one bit set to 1 is UNPREDICTABLE. The effect of a write to an unsupported bit is UNPREDICTABLE.

This register resets to the value for the smallest supported parallel trace port size.

This field resets to an IMPLEMENTATION DEFINED value on a Cold reset.

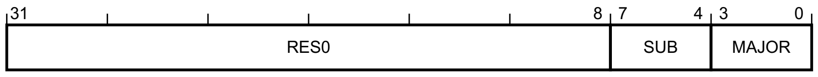
D1.2.198 TPIU_DEVTYPE, TPIU Device Type Register

The TPIU_DEVTYPE characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FCC`. This register is not banked between Security states.

Field descriptions

The TPIU_DEVTYPE bit assignments are:



Bits [31:8]

Reserved, RES0.

SUB, bits [7:4]

Sub-type. Component sub-type.

The possible values of this field are:

`0x0`	Other. Only permitted if the MAJOR field reads as `0x0`.
`0x1`	Trace port. Only permitted if the MAJOR field reads as `0x1`.

This field reads as an IMPLEMENTATION DEFINED value.

MAJOR, bits [3:0]

Major type. Component major type.

The possible values of this field are:

`0x0`	Miscellaneous.
`0x1`	Trace sink.

This field reads as an IMPLEMENTATION DEFINED value.

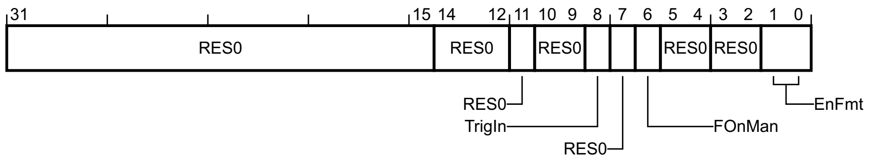
D1.2.199 TPIU_FFCR, TPIU Formatter and Flush Control Register

The TPIU_FFCR characteristics are:

Purpose	Controls the TPIU formatter. This register might contain other formatter and flush control fields that are outside the scope of the architecture. Contact Arm for more information.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
	If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the TPIU is implemented.
	This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read/write register located at `0xE0040304`.
	This register is not banked between Security states.

Field descriptions

The TPIU_FFCR bit assignments are:



Bits [31:15,11,7,3:2]

Reserved, RES0.

Bits [14:12]

Reserved for formatter stop controls.

Reserved, RES0.

Bits [10:9]

Reserved for additional trigger mark controls.

Reserved, RES0.

TrigIn, bit [8]

Trigger input asserted. Indicate a trigger on the trace port when an IMPLEMENTATION DEFINED TRIGIN signal is asserted.

It is IMPLEMENTATION DEFINED whether this bit is R/W or RAO.

This bit resets to zero on a Cold reset.

FOnMan, bit [6]

Flush On Manual. Setting this bit to 1 generates a flush. The TPIU clears the bit to 0 when the flush completes.

This bit resets to zero on a Cold reset.

Bits [5:4]

Reserved for additional flush controls.

Reserved, RES0.

EnFmt, bits [1:0]

Formatter control. Selects the output formatting mode.

The possible values of this field are:

`0b00`	Bypass. Disable formatting. Only supported when SWO mode is selected. Only a single trace source is supported in bypass mode: • If only a single trace source is connected to this TPIU, it is selected. • If multiple sources (including the ITM) are implemented and connected to this TPIU, then all other trace sources, except for the ITM, must be disabled. Otherwise, the trace output is UNPREDICTABLE. All other trace sources are discarded.
`0b10`	Continuous. Enable formatting and embed triggers and null cycles in the formatted output.

All other values are reserved.

If no formatter is implemented, this field is RES0. This field must be set to `0b10` when the parallel trace port is selected, or when using multiple trace sources. Changing the value of this field when TPIU_FFSR.FtStopped is 0 is UNPREDICTABLE.

This field resets to zero on a Cold reset.

Note

An optional TRACECTL pin might be implemented as part of the parallel trace port that allows Bypass mode when using a parallel trace port and a further mode, EnFmt == `0b01`. The CoreSight architecture describes EnFmt[1] as the EnFCont bit and EnFmt[0] as the EnFTC bit.

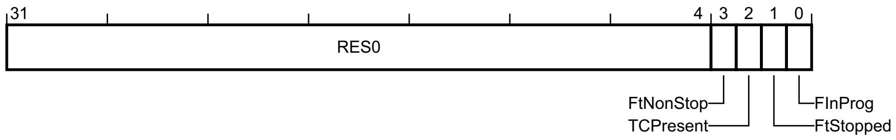
D1.2.200 TPIU_FFSR, TPIU Formatter and Flush Status Register

The TPIU_FFSR characteristics are:

Purpose	Shows the status and capabilities of the TPIU formatter.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault.
	This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.
	If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the TPIU is implemented.
	This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040300`.
	This register is not banked between Security states.

Field descriptions

The TPIU_FFSR bit assignments are:



Bits [31:4]

Reserved, RES0.

FtNonStop, bit [3]

Non-stop formatter. Indicates the formatter cannot be stopped.

The possible values of this bit are:

0	Formatter can be stopped.
1	Formatter cannot be stopped.

If no formatter is implemented, this bit is RAO.

TCPresent, bit [2]

TRACECTL present. Indicates presence of the TRACECTL pin.

The possible values of this bit are:

0	No TRACECTL pin is available. The data formatter must be used and only in continuous mode.
1	The optional TRACECTL pin is present.

If a parallel trace port is not implemented, this bit is RAZ.

Note

If a parallel trace port is implemented, Arm recommends the TRACECTL pin is not implemented.

FtStopped, bit [1]

Formatter stopped. Indicates the formatter is stopped.

The possible values of this bit are:

0	Formatter is enabled.

1	The formatter has received a stop request signal and all trace data and post-amble has been output. Any further trace data is ignored.

If no formatter is implemented, or the formatter cannot be stopped, this bit is RAZ.

FInProg, bit [0]

Flush in progress. Set to 1 when a flush is initiated and clears to zero when all data received before the flush is acknowledged has been output on the trace port. That is, the trace has been received at the sink, formatted, and output on the trace port.

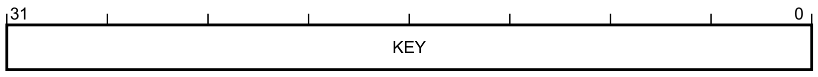
D1.2.201 TPIU_LAR, TPIU Software Lock Access Register

The TPIU_LAR characteristics are:

Purpose	Provides CoreSight Software Lock control for the TPIU, see the *ARM® CoreSight™ Architecture Specification* for details.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. This register is RAZ/WI if accessed via the debugger.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented. Present only if the optional Software Lock is implemented. This register is RAZ/WI if the Software Lock is not implemented.
Attributes	32-bit write-only register located at `0xE0040FB0`. This register is not banked between Security states.

Field descriptions

The TPIU_LAR bit assignments are:



KEY, bits [31:0]

Lock Access control.

Writing the key value `0xC5ACCE55` to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

D1.2.202 TPIU_LSR, TPIU Software Lock Status Register

The TPIU_LSR characteristics are:

Purpose Provides CoreSight Software Lock status information for the TPIU, see the *ARM® CoreSight™ Architecture Specification* for details.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

This register is RAZ/WI if accessed via the debugger.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

This register is RES0 if the TPIU is not implemented.

Present only if the optional Software Lock is implemented.

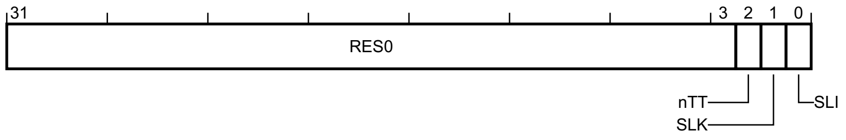
This register is RAZ/WI if the Software Lock is not implemented.

Attributes 32-bit read-only register located at `0xE0040FB4`.

This register is not banked between Security states.

Field descriptions

The TPIU_LSR bit assignments are:



Bits [31:3]

Reserved, RES0.

nTT, bit [2]

Not thirty-two bit. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as zero.

SLK, bit [1]

Software Lock status. See the *ARM® CoreSight™ Architecture Specification*.

The possible values of this bit are:

0	Lock clear. Software writes are permitted to this component's registers.
1	Lock set. Software writes to this component's registers 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.

D1.2.203 TPIU_PIDR0, TPIU Peripheral Identification Register 0

The TPIU_PIDR0 characteristics are:

Purpose Provides CoreSight discovery information for the TPIU.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

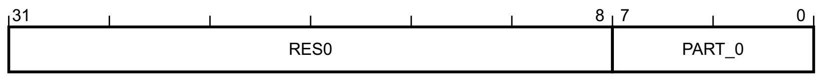
This register is RES0 if the TPIU is not implemented.

Attributes 32-bit read-only register located at `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.

D1.2.204 TPIU_PIDR1, TPIU Peripheral Identification Register 1

The TPIU_PIDR1 characteristics are:

Purpose Provides CoreSight discovery information for the TPIU.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

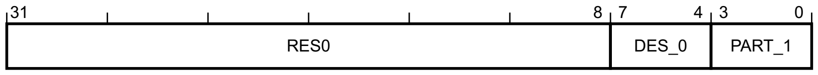
This register is RES0 if the TPIU is not implemented.

Attributes 32-bit read-only register located at `0xE0040FE4`.

This register is not banked between Security states.

Field descriptions

The TPIU_PIDR1 bit assignments are:



Bits [31:8]

Reserved, RES0.

DES_0, bits [7:4]

JEP106 identification code bits [3:0]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

PART_1, bits [3:0]

Part number bits [11:8]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

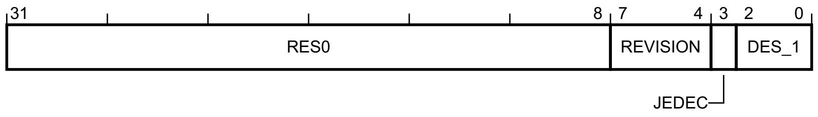
D1.2.205 TPIU_PIDR2, TPIU Peripheral Identification Register 2

The TPIU_PIDR2 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FE8`. This register is not banked between Security states.

Field descriptions

The TPIU_PIDR2 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVISION, bits [7:4]

Component revision. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

JEDEC, bit [3]

JEDEC assignee value is used. See the *ARM® CoreSight™ Architecture Specification*.

This bit reads as one.

DES_1, bits [2:0]

JEP106 identification code bits [6:4]. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

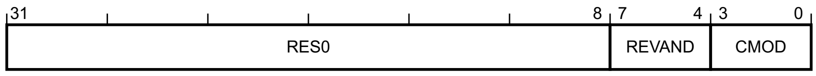
D1.2.206 TPIU_PIDR3, TPIU Peripheral Identification Register 3

The TPIU_PIDR3 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FEC`. This register is not banked between Security states.

Field descriptions

The TPIU_PIDR3 bit assignments are:



Bits [31:8]

Reserved, RES0.

REVAND, bits [7:4]

RevAnd. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

CMOD, bits [3:0]

Customer Modified. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

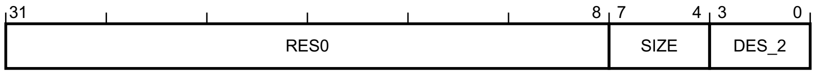
D1.2.207 TPIU_PIDR4, TPIU Peripheral Identification Register 4

The TPIU_PIDR4 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FD0`. This register is not banked between Security states.

Field descriptions

The TPIU_PIDR4 bit assignments are:



Bits [31:8]

Reserved, RES0.

SIZE, bits [7:4]

4KB count. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as zero.

DES_2, bits [3:0]

JEP106 continuation code. See the *ARM® CoreSight™ Architecture Specification*.

This field reads as an IMPLEMENTATION DEFINED value.

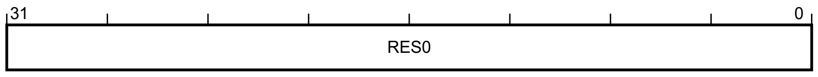
D1.2.208 TPIU_PIDR5, TPIU Peripheral Identification Register 5

The TPIU_PIDR5 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FD4`. This register is not banked between Security states.

Field descriptions

The TPIU_PIDR5 bit assignments are:



Bits [31:0]

Reserved, RES0.

D1.2.209 TPIU_PIDR6, TPIU Peripheral Identification Register 6

The TPIU_PIDR6 characteristics are:

Purpose Provides CoreSight discovery information for the TPIU.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if CoreSight identification is implemented.

This register is RES0 if CoreSight identification is not implemented.

Present only if the TPIU is implemented.

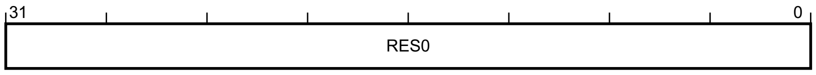
This register is RES0 if the TPIU is not implemented.

Attributes 32-bit read-only register located at `0xE0040FD8`.

This register is not banked between Security states.

Field descriptions

The TPIU_PIDR6 bit assignments are:



Bits [31:0]

Reserved, RES0.

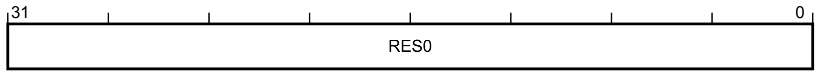
D1.2.210 TPIU_PIDR7, TPIU Peripheral Identification Register 7

The TPIU_PIDR7 characteristics are:

Purpose	Provides CoreSight discovery information for the TPIU.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if CoreSight identification is implemented. This register is RES0 if CoreSight identification is not implemented. Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040FDC`. This register is not banked between Security states.

Field descriptions

The TPIU_PIDR7 bit assignments are:



Bits [31:0]

Reserved, RES0.

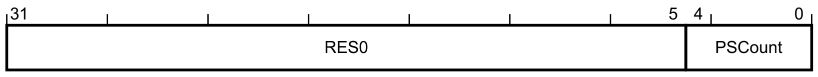
D1.2.211 TPIU_PSCR, TPIU Periodic Synchronization Control Register

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. 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.

D1.2.212 TPIU_SPPR, TPIU Selected Pin Protocol Register

The TPIU_SPPR characteristics are:

Purpose Selects the protocol used for trace output.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

If a debugger changes the register value while the TPIU is transmitting data, the effect on the output stream is UNPREDICTABLE and the required recovery process is IMPLEMENTATION DEFINED.

Configurations Present only if the TPIU is implemented and supports SWO.

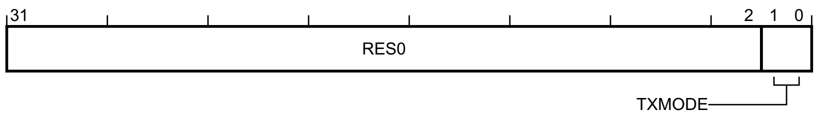
This register is RES0 if the TPIU is not implemented or does not support SWO.

Attributes 32-bit read/write register located at `0xE00400F0`.

This register is not banked between Security states.

Field descriptions

The TPIU_SPPR bit assignments are:



Bits [31:2]

Reserved, RES0.

TXMODE, bits [1:0]

Transmit mode. Specifies the protocol for trace output from the TPIU.

The possible values of this field are:

`0b00`	Parallel trace port mode. This value is reserved if TPIU_TYPE.PTINVALID == 1.
`0b01`	Asynchronous SWO, using Manchester encoding. This value is reserved if TPIU_TYPE.MANCVALID == 0.
`0b10`	Asynchronous SWO, using NRZ encoding. This value is reserved if TPIU_TYPE.NRZVALID == 0.

All other values are reserved.

The effect of selecting a reserved value, or a mode that the implementation does not support, is UNPREDICTABLE.

This field resets to an IMPLEMENTATION DEFINED value on a Cold reset.

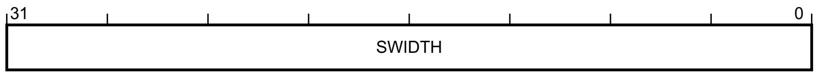
D1.2.213 TPIU_SSPSR, TPIU Supported Parallel Port Sizes Register

The TPIU_SSPSR characteristics are:

Purpose	Indicates the supported parallel trace port sizes.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE. If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.
Configurations	Present only if the TPIU is implemented. This register is RES0 if the TPIU is not implemented.
Attributes	32-bit read-only register located at `0xE0040000`. This register is not banked between Security states.

Field descriptions

The TPIU_SSPSR bit assignments are:



SWIDTH, bits [31:0]

Supported width. SWIDTH[*m*] indicates whether a parallel trace port width of (*m*+1) is supported.

The possible values of each bit are:

0	Parallel trace port width (*m*+1) not supported.
1	Parallel trace port width (*m*+1) supported.

The value of this register is IMPLEMENTATION DEFINED.

This field reads as an IMPLEMENTATION DEFINED value.

D1.2.214 TPIU_TYPE, TPIU Device Identifier Register

The TPIU_TYPE characteristics are:

Purpose Describes the TPIU to a debugger.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

If the Main Extension is not implemented then it is IMPLEMENTATION DEFINED whether this register is accessible only to the debugger and RES0 for software. Otherwise the register is accessible to the debugger and software.

Configurations Present only if the TPIU is implemented.

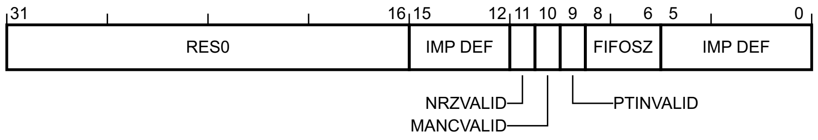
This register is RES0 if the TPIU is not implemented.

Attributes 32-bit read-only register located at `0xE0040FC8`.

This register is not banked between Security states.

Field descriptions

The TPIU_TYPE bit assignments are:



Bits [31:16]

Reserved, RES0.

Bits [15:12]

IMPLEMENTATION DEFINED.

NRZVALID, bit [11]

NRZ valid. Indicates support for SWO using UART/NRZ encoding.

The possible values of this bit are:

0	Not supported.
1	Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

MANCVALID, bit [10]

Manchester valid. Indicates support for SWO using Manchester encoding.

The possible values of this bit are:

0	Not supported.
1	Supported.

This bit reads as an IMPLEMENTATION DEFINED value.

PTINVALID, bit [9]

Parallel Trace Interface invalid. Indicates support for parallel trace port operation.

The possible values of this bit are:

0	Supported.
1	Not supported.

This bit reads as an IMPLEMENTATION DEFINED value.

FIFOSZ, bits [8:6]

FIFO depth. Indicates the minimum implemented size of the TPIU output FIFO for trace data.

The possible values of this field are:

0	IMPLEMENTATION DEFINED FIFO depth.
Other	Minimum FIFO size is 2^{FIFOSZ}.

For example, a value of `0b011` indicates a FIFO size of at least $2^3 = 8$ bytes.

This field reads as an IMPLEMENTATION DEFINED value.

Bits [5:0]

IMPLEMENTATION DEFINED.

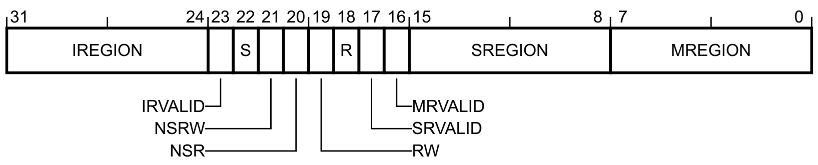
D1.2.215 TT_RESP, Test Target Response Payload

The TT_RESP characteristics are:

Purpose	Provides the response information from a TT, TTA, TTT, or TTAT instruction.
Usage constraints	None.
Configurations	All.
Attributes	32-bit payload.

Field descriptions

The TT_RESP bit assignments are:



IREGION, bits [31:24]

IDAU region number. Indicates the IDAU region number containing the target address.

This field is zero if IRVALID is zero.

IRVALID, bit [23]

IREGION valid flag. For a Secure request, indicates the validity of the IREGION field.

The possible values of this bit are:

0	IREGION content not valid.
1	IREGION content valid.

This bit is always zero if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction was executed from the Non-secure state.

S, bit [22]

Security. For a Secure request, indicates the Security attribute of the target address.

The possible values of this bit are:

0	Target address is Non-secure.
1	Target address is Secure.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

NSRW, bit [21]

Non-secure read and writable. Equal to RW AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the RW field is valid.

NSR, bit [20]

Non-secure readable. Equal to R AND NOT S. Can be used in combination with the LSLS (immediate) instruction to check both the MPU and SAU/IDAU permissions. This field is only valid if the instruction was executed from Secure state and the R field is valid.

RW, bit [19]

Read and writable.

Set to 1 if the memory location can be read and written according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

R, bit [18]

Readable.

Read accessibility. Set to 1 if the memory location can be read according to the permissions of the selected MPU when operating in the current mode. For TTT and TTAT, this field returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.

This field is invalid and RAZ if the TT instruction was executed from an unprivileged mode and the A flag was not specified. This field is also RAZ if the address matches multiple MPU regions.

SRVALID, bit [17]

SREGION valid flag. For a Secure request indicates validity of the SREGION field.

The possible values of this bit are:

0	SREGION content not valid.
1	SREGION content valid.

This bit is always zero if the requesting TT instruction was executed from the Non-secure state.

The SREGION field will be invalid if any of the following are true:

- SAU_CTRL.ENABLE is set to zero.
- The address specified in the SREGION field does not match any 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 will be invalid if any of the following is true:

- The MPU is not implemented or MPU_CTRL.ENABLE is set to zero.
- The address specified by the MREGION field does not match any enabled MPU regions.
- The address matched multiple MPU regions.
- The TT instruction was executed from an unprivileged mode and the A flag was not specified.

SREGION, bits [15:8]

SAU region number. Holds the SAU region that the address maps to.

This field is only valid if the instruction was executed from Secure state. This field is zero if SRVALID is 0.

MREGION, bits [7:0]

MPU region number. Holds the MPU region that the address maps to.

This field is zero if MRVALID is 0.

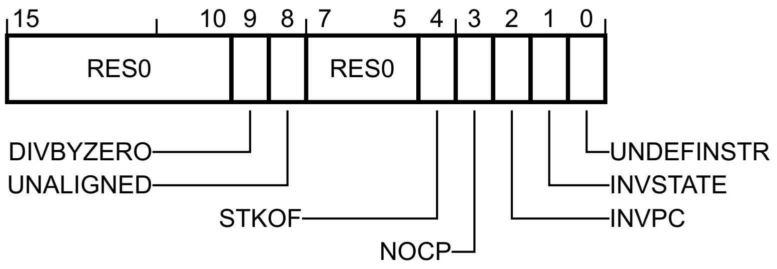
D1.2.216 UFSR, UsageFault Status Register

The UFSR characteristics are:

Purpose	Contains the status for some instruction execution faults, and for data access faults.
Usage constraints	Privileged access permitted only. Unprivileged accesses generate a fault. If the Main Extension is implemented, this register is word, halfword, and byte accessible. If the Main Extension is not implemented, this register is word accessible only, halfword and byte accesses are UNPREDICTABLE.
Configurations	Present only if the Main Extension is implemented. This register is RES0 if the Main Extension is not implemented.
Attributes	16-bit read/write-one-to-clear register located at `0xE000ED2A`. Secure software can access the Non-secure version of this register via UFSR_NS located at `0xE002ED2A`. The location `0xE002ED2A` is RES0 to software executing in Non-secure state and the debugger. This register is banked between Security states. This register is part of CFSR.

Field descriptions

The UFSR bit assignments are:



Bits [15:10]

Reserved, RES0.

DIVBYZERO, bit [9]

Divide by zero flag. Sticky flag indicating whether an integer division by zero error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

UNALIGNED, bit [8]

Unaligned access flag. Sticky flag indicating whether an unaligned access error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

Bits [7:5]

Reserved, RES0.

STKOF, bit [4]

Stack overflow flag. Sticky flag indicating whether a stack overflow error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

NOCP, bit [3]

No coprocessor flag. Sticky flag indicating whether a coprocessor disabled or not present error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

INVPC, bit [2]

Invalid PC flag. Sticky flag indicating whether an integrity check error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

INVSTATE, bit [1]

Invalid state flag. Sticky flag indicating whether an EPSR.T or EPSR.IT validity error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This bit resets to zero on a Warm reset.

UNDEFINSTR, bit [0]

UNDEFINED instruction flag. Sticky flag indicating whether an UNDEFINED instruction error has occurred.

The possible values of this bit are:

0	Error has not occurred.
1	Error has occurred.

This includes attempting to execute an UNDEFINED instruction associated with an enable coprocessor.

This bit resets to zero on a Warm reset.

D1.2.217 VTOR, Vector Table Offset Register

The VTOR characteristics are:

Purpose Holds the vector table address for the selected Security state.

Usage constraints Privileged access permitted only. Unprivileged accesses generate a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

Configurations This register is always implemented.

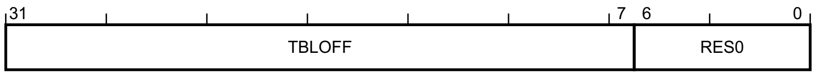
Attributes To allow lock down of this register it is IMPLEMENTATION DEFINED whether this register is writable.

Secure software can access the Non-secure version of this register via VTOR_NS located at `0xE002ED08`. The location `0xE002ED08` is RES0 to software executing in Non-secure state and the debugger.

This register is banked between Security states.

Field descriptions

The VTOR bit assignments are:



TBLOFF, bits [31:7]

Table offset. Bits [31:7] of the vector table address for the selected Security state.

It is IMPLEMENTATION DEFINED whether any of the TBLOFF bits are WI.

This field resets to an IMPLEMENTATION DEFINED value on a Warm reset.

Bits [6:0]

Reserved, RES0.

D1.2.218 XPSR, Combined Program Status Registers

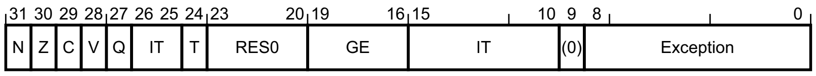
The XPSR characteristics are:

Purpose	Provides access to a combination of the APSR, EPSR and IPSR.
Usage constraints	Privileged access only. Unprivileged access is RAZ/WI, unless otherwise stated.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is not banked between Security states.

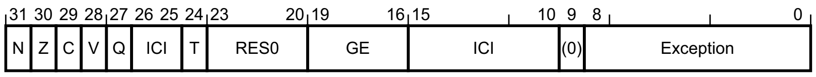
Field descriptions

The XPSR bit assignments are:

When `{XPSR[26:25], XPSR[11:10]} != 0`:



When `{XPSR[26:25], XPSR[11:10]} == 0`:



N, bit [31]

Negative flag. Reads or writes the current value of APSR.N.

Z, bit [30]

Zero flag. Reads or writes the current value of APSR.Z.

C, bit [29]

Carry flag. Reads or writes the current value of APSR.C.

V, bit [28]

Overflow flag. Reads or writes the current value of APSR.V.

Q, bit [27]

Saturate flag. Reads or writes the current value of APSR.Q.

T, bit [24]

T32 state. Reads or writes the current value of EPSR.T.

Bits [23:20]

Reserved, RES0.

GE, bits [19:16]

Greater-than or equal flag. Reads or writes the current value of APSR.GE.

IT, bits [15:10,26:25], when {XPSR[26:25], XPSR[11:10]} != 0

If-then flags. Reads or writes the current value of EPSR.IT.

ICI, bits [26:25,15:10], when {XPSR[26:25], XPSR[11:10]} == 0

Interrupt continuation flags. Reads or writes the current value of EPSR.ICI.

Bit [9]

Reserved, RES0.

Exception, bits [8:0]

Exception number. Reads or writes the current value of IPSR.Exception.

Part E
Armv8-M Pseudocode

Chapter E1
Arm Pseudocode Definition

This chapter provides a definition of the pseudocode that is used in this manual, and defines some *built-in* functions that are used by pseudocode. 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:

- *Data types* on page E1-1187.
- *Operators* on page E1-1192.
- *Statements and control structures* on page E1-1198.

Built-in functions on page E1-1204 describes some built-in functions that are used by the pseudocode functions that are described elsewhere in this manual. *Arm pseudocode definition index* on page E1-1207 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 is described by a particular piece of pseudocode.

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `UNDEFINED`, `CONSTRAINED_UNPREDICTABLE` and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior that is indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior that is indicated by the pseudocode occurs.

For more information, see *Special statements* on page E1-1202.

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 type of a literal is determined by its syntax. 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`.

`<register_name>_Type` and `<payload>_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.

```
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 *Function and procedure calls* on page E1-1198.

E1.3 Operators

This section describes:

- *Relational operators*.
- *Boolean operators*.
- *Bitstring operators* on page E1-1193.
- *Arithmetic operators* on page E1-1193.
- *The assignment operator* on page E1-1194.
- *Precedence rules* on page E1-1196.
- *Conditional expressions* on page E1-1196.
- *Operator polymorphism* on page E1-1196.

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.

Set membership with `IN`

`<expression> IN {<set>}` produces `TRUE` if `<expression>` is a member of `<set>`. Otherwise, it is `FALSE`. `<set>` must be a list of expressions that are separated by commas.

E1.3.2 Boolean operators

If `x` is a boolean expression, then `!x` is its logical inverse.

If `x` and `y` are boolean expressions, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

Note

This is known as short circuit evaluation.

If `x` and `y` are `booleans`, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

Note

If x and y are booleans or boolean expressions, then the result of x != y is the same as the result of exclusive-ORing x and y together. The operator EOR only accepts bitstring arguments.

E1.3.3 Bitstring operators

The following operations can be applied only to bitstrings.

Logical operations on bitstrings

If x is a bitstring, NOT(x) is the bitstring of the same length that is obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, x 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:

- >= 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,xn 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 *Boolean operators* on page E1-1192.
2. Operators on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.
3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but do not need to be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j` and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

E1.3.7 Conditional expressions

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

E1.3.8 Operator polymorphism

Operators in pseudocode can be polymorphic, with different functionality when applied to different data types. Each resulting form of an operator has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals and bitstrings.

Table E1-1 summarizes the operand types valid for each unary operator and the result type. Table E1-2 summarizes the operand types valid for each binary operator and the result type.

Table E1-1 Result and operand types that are permitted for unary operators

Operator	Operand Type	Result Type
-	integer	integer
	real	real
NOT	bits(N)	bits(N)
!	boolean	boolean

Table E1-2 Result and operand types that are permitted for binary operators

Operator	First operand type	Second operand type	Result type
==	bits(N)	integer	boolean
		bits(N)	
	integer	integer	
	real	real	
	enumeration	enumeration	
	boolean	boolean	

Table E1-2 Result and operand types that are permitted for binary operators (continued)

Operator	First operand type	Second operand type	Result type
`!=`	`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)`
		`integer`	
`<<, >>`	`integer`	`integer`	`integer`
`*`	`integer`	`integer`	`integer`
	`real`	`real`	`real`
	`bits(N)`	`bits(N)`	`bits(N)`
`/`	`real`	`real`	`real`
`DIV`	`integer`	`integer`	`integer`
`MOD`	`integer`	`integer`	`integer`
	`bits(N)`	`integer`	
`&&, ||`	`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 semicolons. 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* on page E1-1192.

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>`.

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 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 that the Undefined Instruction exception is taken.

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.

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

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 that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

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:

```
IMPLEMENTATION_DEFINED {"<text>"};
```

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 IMPLEMENTATION DEFINED. 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:

- *Bitstring manipulation functions*.
- *Arithmetic functions* on page E1-1205.

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:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
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 two's complement representation is x.

UInt() is the integer whose unsigned representation is x.

Int(x, unsigned) returns either SInt(x) or UInt(x) depending on the value of its second argument.

E1.5.2 Arithmetic functions

This section defines built-in arithmetic functions.

Absolute value

If x is either of type real or integer, Abs(x) returns the absolute value of x. The result is the same type as x.

Rounding and aligning

If `x` is a real:

- `RoundDown(x)` produces the largest integer `n` so that `n <= 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 on page E1-1208 which contains the pseudocode keywords and control structures.
- Table E1-6 on page E1-1209 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
	Extract named bit or field from a register
`:`	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
`/`	Division of integers and reals (real result)

Table E1-4 Index of pseudocode operators (continued)

Operator	Meaning
`&&`	Boolean AND
`<`	*Less than* comparison of integers and reals
`<…>`	Slicing of specified bits of bitstring or integer
`<<`	Multiply 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`	Tests 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 case in `case … of …` control structure

Table E1-5 Index of pseudocode keywords and control structures (continued)

Operator	Meaning
`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`)
`when`	Introduces 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
`UNPREDICTABLE`	Unspecified behavior

E1.7 Additional functions

The following functions are not listed in Chapter 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…* on page E1-1203.

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* on page E1-1202.

Chapter E2
Pseudocode Specification

This chapter specifies the ARMv8-M pseudocode. It contains the following section:

E2.1 Alphabetical Pseudocode List

E2.1.1 ALUWritePC

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

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

E2.1.2 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;
```

E2.1.3 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);
```

E2.1.4 AccType

```
// Memory reference access type
enumeration AccType { AccType_NORMAL,      // Normal loads and stores
                      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_VECTABLE     // Vector table fetch
                    };
```

E2.1.5 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
)
```

E2.1.6 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 bits cleared
    // PRIMASK, FAULTMASK, BASEPRI unchanged on exception entry
    if HaveFPExt() 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);
```

E2.1.7 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);
```

E2.1.8 AddressDescriptor

```
// Descriptor used to access the underlying memory array

type AddressDescriptor is (
    MemoryAttributes memattrs,
    bits(32)         paddress,   // Physical Address
    AccessAttributes accattrs
)
```

E2.1.9 BKPTInstrDebugEvent

```
// BKPTInstrDebugEvent()
// =====================
// Generates a debug event based on BKPT Instruction.

BKPTInstrDebugEvent()
    if !GenerateDebugEventResponse() then
        excInfo = CreateException(HardFault, FALSE, boolean UNKNOWN);
        HandleException(excInfo);
```

E2.1.10 BLXWritePC

```
// BLXWritePC()
// ============

BLXWritePC(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
         BranchToNS(address);
    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');
```

E2.1.11 BXWritePC

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

ExcInfo BXWritePC(bits(32) address, boolean allowNonSecure)
    exc = DefaultExcInfo();
    if HaveSecurityExt() && address == '1111 1110 1111 1111 1111 1111 1111 111x' then
        // Unlike exception return, any faults raised during a FNC_RETURN
        // unstacking are raised synchronously with the instruction that triggered
        // the unstacking.
        exc = FunctionReturn();

    elsif CurrentMode() == PEMode_Handler && address<31:24> == '11111111' then
        // 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.
        PendReturnOperation(address);

    elsif HaveSecurityExt() && IsSecure() && allowNonSecure then
        // 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
        BranchToNS(address);

    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');

    return exc;
```

E2.1.12 BigEndian

```
// BigEndian()
// ===========

boolean BigEndian()
    return (AIRCR.ENDIANNESS == '1');
```

E2.1.13 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;
```

E2.1.14 BranchTo

```
// BranchTo()
// ==========

BranchTo(bits(32) address)
    // 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;
    _PCChanged     = TRUE;
    // Clear any pending exception returns
    _PendingReturnOperation = FALSE;
    return;
```

E2.1.15 BranchToAndCommit

```
// BranchToAndCommit()
// ===================

BranchToAndCommit(bits(32) address)
    // This function 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.
    _R[RName_PC]   = address<31:1>:'0';
    _PCChanged     = TRUE;
    _NextInstrAddr = address<31:1>:'0';
    // Clear any pending exception returns
    _PendingReturnOperation = FALSE;
    return;
```

E2.1.16 BranchToNS

```
// BranchToNS()
// ============
// Branch to an address with an option to change from Secure to Non-secure
// state, if currently in Secure state and transition to Non-secure state is allowed.
// Transition to Non-secure state is specified by the LSB bit of target
// address (address<0>).

BranchToNS(bits(32) address)
    assert HaveSecurityExt() && IsSecure();
    EPSR.T = '1';
    if address<0> == '0' then
        CurrentState = SecurityState_NonSecure;
        if HaveFPExt() then CONTROL_S.SFPA = '0';
    BranchTo(address<31:1>:'0');
```

E2.1.17 BranchWritePC

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

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

E2.1.18 CallSupervisor

```
// CallSupervisor()
// ================

CallSupervisor()
    excInfo = CreateException(SVCall, FALSE, boolean UNKNOWN);
    HandleException(excInfo);
```

E2.1.19 CanHaltOnEvent

```
// CanHaltOnEvent()
// ================

boolean CanHaltOnEvent(boolean is_secure)
    if !HaveSecurityExt() then assert !is_secure;
    return (HaveHaltingDebug() && HaltingDebugAllowed() && DHCSR.C_DEBUGEN == '1' &&
            DHCSR.S_HALT == '0' && (!is_secure || DHCSR.S_SDE == '1'));
```

E2.1.20 CanPendMonitorOnEvent

```
// CanPendMonitorOnEvent()
// =======================

boolean CanPendMonitorOnEvent(boolean is_secure, boolean check_pri)
    if !HaveSecurityExt() then assert !is_secure;
    return (HaveDebugMonitor() && !CanHaltOnEvent(is_secure) && DEMCR.MON_EN == '1' &&
            DHCSR.S_HALT == '0' && (!is_secure || DEMCR.SDME == '1') &&
            (!check_pri || ExceptionPriority(DebugMonitor, is_secure, TRUE) < ExecutionPriority()));
```

E2.1.21 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());
```

E2.1.22 CheckDecodeFaults

```
// CheckDecodeFaults()
// ===================
// Check and raise faults in the correct order

CheckDecodeFaults(boolean dp_operation)
    // Check FP Extension is supported, else raise NOCP Fault
    if !HaveFPExt() then
        secure = IsSecure();
        if secure then
            UFSR_S.NOCP  = '1';
        else
            UFSR_NS.NOCP = '1';
        excInfo = CreateException(UsageFault, TRUE, secure);
        HandleException(excInfo);

    // Check access to FP coprocessor is enabled, else raise NOCP Fault
    excInfo = CheckCPEnabled(10);
    HandleException(excInfo);

    if dp_operation && HaveSPFPOnly() then UNDEFINED;


CheckDecodeFaults()
    CheckDecodeFaults(FALSE);
```

E2.1.23 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_ORDERED
                    fsr.MMARVALID = '1';
                    fsr.DACCVIOL  = '1';
                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;
```

E2.1.24 ClearEventRegister

```
// ClearEventRegister
// ==================
// Clears the Event register

ClearEventRegister();
```

E2.1.25 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);
```

E2.1.26 ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====================
// Clear local exclusive monitor records for the PE.

ClearExclusiveLocal(integer processorid);
```

E2.1.27 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);
```

E2.1.28 ConditionHolds

```
// ConditionHolds()
// ================

boolean ConditionHolds(bits(4) cond)

    // Evaluate base condition.
    case cond<3:1> of
        when '000'  result = (APSR.Z == '1');                         // EQ or NE
        when '001'  result = (APSR.C == '1');                         // CS or CC
        when '010'  result = (APSR.N == '1');                         // MI or PL
        when '011'  result = (APSR.V == '1');                         // VS or VC
        when '100'  result = (APSR.C == '1') && (APSR.Z == '0');      // HI or LS
        when '101'  result = (APSR.N == APSR.V);                      // GE or LT
        when '110'  result = (APSR.N == APSR.V) && (APSR.Z == '0');  // GT or LE
        when '111'  result = TRUE;                                    // AL

    // Condition flag values in the set '111x' indicate the instruction is always executed.
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

E2.1.29 ConditionPassed

```
// ConditionPassed()
// =================

boolean ConditionPassed()
    return ConditionHolds(CurrentCond());
```

E2.1.30 ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================
// This is a wrapper for UNPREDICTABLE cases where the constrained result is
// either TRUE or FALSE.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

E2.1.31 ConsumeExcStackFrame

```
// ConsumeExcStackFrame()
// ======================

ConsumeExcStackFrame(EXC_RETURN_Type excReturn, bit fourByteAlign)
    // Calculate the size of the integer part of the stack frame
    toSecure = HaveSecurityExt() && excReturn<6> == '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 HaveFPExt() && 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);
    _R[spName] = (_SP(spName) + framesize) OR ZeroExtend(fourByteAlign:'00',32);
```

E2.1.32 Coproc_Accepted

```
// Coproc_Accepted
// ===============
// Check whether a coprocessor accepts an instruction.

boolean Coproc_Accepted(integer cp_num, bits(32) instr);
```

E2.1.33 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);
```

E2.1.34 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);
```

E2.1.35 Coproc_GetOneWord

```
// Coproc_GetOneWord
// =================
// Gets the 32-bit word for an MRC instruction from the coprocessor

bits(32) Coproc_GetOneWord(integer cp_num, bits(32) instr);
```

E2.1.36 Coproc_GetTwoWords

```
// Coproc_GetTwoWords
// ==================
// Get two 32-bit words for an MRRC instruction from the coprocessor

(bits(32), bits(32)) Coproc_GetTwoWords(integer cp_num, bits(32) instr);
```

E2.1.37 Coproc_GetWordToStore

```
// Coproc_GetWordToStore
// =====================
// Gets the next 32-bit word to store for an STC instruction from the coprocessor

bits(32) Coproc_GetWordToStore(integer cp_num, bits(32) instr);
```

E2.1.38 Coproc_InternalOperation

```
// Coproc_InternalOperation
// ========================
// Instructs a coprocessor to perform the internal operation requested
// by a CDP instruction

Coproc_InternalOperation(integer cp_num, bits(32) instr);
```

E2.1.39 Coproc_SendLoadedWord

```
// Coproc_SendLoadedWord
// =====================
// Sends a loaded 32-bit word for an LDC instruction to the coprocessor

Coproc_SendLoadedWord(bits(32) word, integer cp_num, bits(32) instr);
```

E2.1.40 Coproc_SendOneWord

```
// Coproc_SendOneWord
// ==================
// Sends the 32-bit word for an MCR instruction to the coprocessor

Coproc_SendOneWord(bits(32) word, integer cp_num, bits(32) instr);
```

E2.1.41 Coproc_SendTwoWords

```
// Coproc_SendTwoWords
// ===================
// Send two 32-bit words for an MCRR instruction to the coprocessor.

Coproc_SendTwoWords(bits(32) word2, bits(32) word1, integer cp_num, bits(32) instr);
```

E2.1.42 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);
```

E2.1.43 CurrentCond

```
// CurrentCond()
// =============
// Returns condition specifier of current instruction.

bits(4) CurrentCond();
```

E2.1.44 CurrentMode

```
// CurrentMode()
// =============

PEMode CurrentMode()
    return if IPSR == NoFault then PEMode_Thread else PEMode_Handler;
```

E2.1.45 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');
```

E2.1.46 D

```
// D[]
// ===

// Non-assignment form

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    return _D[n];

// Assignment form

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    if n >= 16 && VFPSmallRegisterBank() then UNDEFINED;
    _D[n] = value;
    return;
```

E2.1.47 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);
```

E2.1.48 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;
```

E2.1.49 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;
```

E2.1.50 DWT_DataMatch

```
// DWT_DataMatch()
// ===============
// Perform varioius 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;
```

E2.1.51 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
            dmask = '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)  dmask<0> = '1';
                    when (1,2)  dmask<UInt(DWT_COMP[N-1]<0>)> = '1';
                    when (1,4)  dmask<UInt(DWT_COMP[N-1]<1:0>)> = '1';
                    when (2,2)  dmask<1:0> = '11';
                    when (2,4)
                        dmask<UInt(DWT_COMP[N-1]<1:0>)+1:UInt(DWT_COMP[N-1]<1:0>)> = '11';
                    when (4,4)  dmask = '1111';
                    otherwise   dmask = '0000';     // vsize > dsize: no match
        else
            case dsize of
                when 1  dmask = '0001';
                when 2  dmask = '0011';
                when 4  dmask = '1111';

        // 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 = dmask<3> == '1' && D == d;
        C_c = dmask<2> == '1' && C == c;
        B_b = dmask<1> == '1' && B == b;
        A_a = dmask<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;

        match = match_data && match_lsc;
    else
        match = FALSE;

    return match;
```

E2.1.52 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;
```

E2.1.53 DWT_InstructionMatch

```
// DWT_InstructionMatch()
// ====================
// Perform varioius 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;

    if trigger_debug_event then
        debug_event = SetDWTDebugEvent(IsSecure());
    return;
```

E2.1.54 DWT_ValidMatch

```
// DWT_ValidMatch()
// ================
// Returns TRUE if this match is permitted by the current authentication controls, FALSE otherwise.

boolean DWT_ValidMatch(integer N, boolean secure_match)
    if !HaveSecurityExt() then assert !secure_match;

    // Check for disabled
    if !NoninvasiveDebugAllowed() || DEMCR.TRCENA == '0' || DWT_FUNCTION[N].MATCH == '0000' then
        return FALSE;

    // Check for Debug event
    if DWT_FUNCTION[N].ACTION == '01' then
        hlt_en = CanHaltOnEvent(secure_match);
        // Ignore priority when checking whether DebugMonitor activates DWT matches
        mon_en = HaveDebugMonitor() && CanPendMonitorOnEvent(secure_match, FALSE);
        return (hlt_en || mon_en);
    else
        // Otherwise trace or trigger event
        return !secure_match || SecureNoninvasiveDebugAllowed();
```

E2.1.55 DataMemoryBarrier

```
// DataMemoryBarrier()
// ===================
// Perform a Data Memory Barrier operation

DataMemoryBarrier(bits(4) option);
```

E2.1.56 DataSynchronizationBarrier

```
// DataSynchronizationBarrier
// ==========================
// Perform a data synchronization barrier operation

DataSynchronizationBarrier(bits(4) option);
```

E2.1.57 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;
```

E2.1.58 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;
```

E2.1.59 DecodeExecute

```
// DecodeExecute
// =============
// Decode instruction and execute

DecodeExecute(bits(32) instr, bits(32) pc, boolean isT16);
```

E2.1.60 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);
```

E2.1.61 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;
```

E2.1.62 DefaultExcInfo

```
// DefaultExcInfo()
// ================

ExcInfo DefaultExcInfo()
    ExcInfo exc;

    exc.fault       = NoFault;
    exc.origFault   = NoFault;
    exc.isSecure    = TRUE;
    exc.isTerminal  = FALSE;
    exc.inExcTaken  = FALSE;
    exc.lockup      = FALSE;
    exc.termInst    = TRUE;
    return exc;
```

E2.1.63 DefaultMemoryAttributes

```
// DefaultMemoryAttributes()
// =========================

MemoryAttributes DefaultMemoryAttributes(bits(32) address)

    MemoryAttributes memattrs;

    case address<31:29> of
        when '000'
            memattrs.memtype = MemType_Normal;
            memattrs.device = DeviceType UNKNOWN;
            memattrs.innerattrs = '10';
            memattrs.shareable = FALSE;
        when '001'
            memattrs.memtype = MemType_Normal;
            memattrs.device = DeviceType UNKNOWN;
            memattrs.innerattrs = '01';
            memattrs.shareable = FALSE;
        when '010'
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
            memattrs.innerattrs = '00';
            memattrs.shareable = TRUE;
        when '011'
            memattrs.memtype = MemType_Normal;
            memattrs.device = DeviceType UNKNOWN;
            memattrs.innerattrs = '01';
            memattrs.shareable = FALSE;
        when '100'
            memattrs.memtype = MemType_Normal;
            memattrs.device = DeviceType UNKNOWN;
            memattrs.innerattrs = '10';
            memattrs.shareable = FALSE;
        when '101'
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
            memattrs.innerattrs = '00';
            memattrs.shareable = TRUE;
        when '110'
            memattrs.memtype = MemType_Device;
            memattrs.device = DeviceType_nGnRE;
            memattrs.innerattrs = '00';
            memattrs.shareable = TRUE;
        when '111'
            if address<28:20> == '000000000' then
                memattrs.memtype = MemType_Device;
                memattrs.device = DeviceType_nGnRnE;
                memattrs.innerattrs = '00';
                memattrs.shareable = TRUE;
            else
                memattrs.memtype = MemType_Device;
                memattrs.device = DeviceType_nGnRE;
                memattrs.innerattrs = '00';
                memattrs.shareable = TRUE;

    // Outer attributes are the same as the inner attributes in all cases.
    memattrs.outerattrs = memattrs.innerattrs;
    memattrs.outershareable = memattrs.shareable;

    // Setting as UNKNOWN by default. This flag will be  overwritten based on
    // SAU/IDAU checking in SecurityCheck()
    memattrs.NS = boolean UNKNOWN;
    return memattrs;
```

E2.1.64 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;
```

E2.1.65 DerivedLateArrival

```
// DerivedLateArrival()
// ====================

DerivedLateArrival(integer pePriority, integer peNumber, boolean peIsSecure, ExcInfo deInfo,
                   integer oeNumber, boolean oeIsSecure)
    // 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
            ActivateException(oeNumber, oeIsSecure);
            // 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.
            ActivateException(oeNumber, oeIsSecure);
            Lockup(TRUE);
        else
            // DE will be pended below, start execution of the OE
            targetFault    = oeNumber;
            targetIsSecure = oeIsSecure;

    // If not 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 = ExceptionTaken(targetFault, deInfo.inExcTaken, targetIsSecure, TRUE);
    // 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);
```

E2.1.66 DeviceType

```
// Types of memory

enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

E2.1.67 EndOfInstruction

```
// EndOfInstruction
// ================
// Terminates the processing of current instruction.

EndOfInstruction();
```

E2.1.68 EventRegistered

```
// EventRegistered
// ===============
// Returns TRUE if PE Event Register is set to 1 and FALSE otherwise.

boolean EventRegistered();
```

E2.1.69 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.
)
```

E2.1.70 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;
```

E2.1.71 ExceptionDetails

```
// ExceptionDetails()
// ==================

(boolean, boolean) ExceptionDetails(integer exception, boolean isSecure, boolean isSynchronous)
    // Is the exception subject to escalation
    case exception of
        when HardFault
            termInst    = TRUE;
            enabled     = TRUE;
            canEscalate = TRUE;
        when MemManage
            termInst    = TRUE;
            if HaveMainExt() then
                val     = if isSecure then SHCSR_S else SHCSR_NS;
                enabled = val.MEMFAULTENA == '1';
            else
                enabled = FALSE;
            canEscalate = TRUE;
        when BusFault
            termInst    = isSynchronous;
            enabled     = if HaveMainExt()
                          then SHCSR_S.BUSFAULTENA == '1' else FALSE;
            // Async BusFaults only escalate if they are disabled
            canEscalate = termInst || !enabled;
        when UsageFault
            termInst    = TRUE;
            if HaveMainExt() then
                val     = if isSecure then SHCSR_S else SHCSR_NS;
                enabled = val.USGFAULTENA == '1';
            else
                enabled = FALSE;
            canEscalate = TRUE;
        when SecureFault
            termInst    = TRUE;
            enabled     = if HaveMainExt()
                          then SHCSR_S.SECUREFAULTENA == '1' else FALSE;
            canEscalate = TRUE;
        when SVCall
            termInst    = FALSE;
            enabled     = TRUE;
            canEscalate = TRUE;
        when DebugMonitor
            termInst    = TRUE;
            enabled     = if HaveMainExt()
                          then DEMCR.MON_EN == '1' else FALSE;
            canEscalate = FALSE; // TRUE if fault caused by BKPT instruction
        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) || !enabled then
            escalateToHf = TRUE;

    return (escalateToHf, termInst);
```

E2.1.72 ExceptionEnabled

```
// ExceptionEnabled()
// ==================
// Checks whether the given exception is enabled.

boolean ExceptionEnabled(integer exception, boolean secure);
```

E2.1.73 ExceptionEntry

```
// ExceptionEntry()
// ================
// Exception entry is modified according to the behavior of a derived
// exception, see DerivedLateArrival() also.

ExcInfo ExceptionEntry(integer exceptionType, boolean toSecure, boolean instExecOk)

    // PushStack() can abandon memory accesses if a fault occurs during the stacking
    // sequence.
    exc = PushStack(toSecure, instExecOk);
    if exc.fault == NoFault then
        exc = ExceptionTaken(exceptionType, FALSE, toSecure, FALSE);
    return exc;
```

E2.1.74 ExceptionPriority

```
// ExceptionPriority()
// ===================

integer ExceptionPriority(integer n, boolean isSecure, boolean groupPri)
    if HaveMainExt() then
        assert n >= 1 && n <= 511;
    else
        assert n >= 1 && 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;

    // 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;

    return result;
```

E2.1.75 ExceptionReturn

```
// ExceptionReturn()
// =================

(ExcInfo, EXC_RETURN_Type) ExceptionReturn(EXC_RETURN_Type excReturn)
    integer returningExceptionNumber = UInt(IPSR.Exception);

    (exc, excReturn) = ValidateExceptionReturn(excReturn, returningExceptionNumber);
    if exc.fault != NoFault then
        return (exc, excReturn);

    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;

    targetDomainSecure = excReturn.ES == '1';
    DeActivate(returningExceptionNumber, targetDomainSecure);

    // If requested, clear the scratch FP values left in the caller saved
    // registers before returning/tail chaining.
    if HaveFPExt() && FPCCR.CLRONRET == '1' && CONTROL.FPCA == '1' then
        if FPCCR_S.LSPACT == '1' then
            SFSR.LSERR = '1';
            exc = CreateException(SecureFault, TRUE, TRUE);
            return (exc, excReturn);
        else
            for i = 0 to 15
                S[i] = Zeros();
            FPSCR    = Zeros();

    // 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 = TailChain(exception, excIsSecure, excReturn);
            return (exc, excReturn);

    // 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);
```

E2.1.76 ExceptionTaken

```
// ExceptionTaken()
// ================

ExcInfo ExceptionTaken(integer exceptionNumber, boolean doTailChain,
                       boolean excIsSecure, boolean ignStackFaults)
    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() && LR<6> == '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 && LR<0> == '0' then
                LR<5> = '0';
        else                        // Transitioning to Non-secure
            // If the callee registers aren't already on the stack push them now
            if LR<5> == '1' && !(doTailChain && LR<0> == '0') then
                exc = PushCalleeStack(doTailChain);
            // Going to Non-secure exception. Set excReturn.DCRS to default
            // value
            LR<5> = '1';

    // Finalise excReturn value
    if excIsSecure then
        LR<2> = CONTROL_S.SPSEL;
        LR<0> = '1';
    else
        LR<2> = CONTROL_NS.SPSEL;
        LR<0> = '0';

    // Register clearing
    // Caller saved registers: These registers are cleared if exception targets
    // the Non-secure state, otherwise they are UNKNOWN. NOTE: The original
    // values were pushed to the stack.
    callerRegValue = if !HaveSecurityExt() || excIsSecure then bits(32) UNKNOWN else Zeros(32);
    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 excepton 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, of when the exception is Secure and the values have not
    //       been pushed to the stack.
    if HaveSecurityExt() && LR<6> == '1' then
        if excIsSecure then
            if LR<5> == '0' then
                for n = 4 to 11
                    R[n] = bits(32) UNKNOWN;
        else
            for n = 4 to 11
                R[n] = Zeros();

    // If no errors so far (or errors that can be ignored) load the vector address
    if exc.fault == NoFault || ignStackFaults 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 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');
    else
        exc.inExcTaken = TRUE;
    return exc;
```

E2.1.77 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;
```

E2.1.78 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, FALSE, boolean UNKNOWN);
    else
        (excInfo, memaddrdesc) = ValidateAddress(address, AccType_NORMAL,
                                                FindPriv(), IsSecure(), TRUE, TRUE);
    HandleException(excInfo);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    if memaddrdesc.memattrs.shareable then
        passed = passed && IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);
    if passed then
        ClearExclusiveLocal(ProcessorID());
    return passed;
```

E2.1.79 ExecuteCPCheck

```
// ExecuteCPCheck()
// ================

ExecuteCPCheck(integer cp)
    // Check access to coprocessor is enabled
    excInfo = CheckCPEnabled(cp);
    HandleException(excInfo);
```

E2.1.80 ExecuteFPCheck

```
// ExecuteFPCheck()
// ================

ExecuteFPCheck()
    // If FP lazy context save is enabled then save state
    if FPCCR_S.S == '1' then
        lspact = FPCCR_S.LSPACT;
    else
        lspact = FPCCR_NS.LSPACT;
    if lspact == '1' then
        PreserveFPState();

    // Update the ownership of the FP context
    FPCCR_S.S = if IsSecure() then '1' else '0';

    // Update CONTROL.FPCA, and create new FP context
    // if this has been enabled by setting FPCCR.ASPEN to 1
    if FPCCR.ASPEN == '1' &&
       (CONTROL.FPCA == '0' || (IsSecure() && CONTROL_S.SFPA == '0')) then
        FPSCR = FPDSCR<31:0>;
        CONTROL.FPCA = '1';
        if IsSecure() then
            CONTROL_S.SFPA = '1';
    return;
```

E2.1.81 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;

    return priority;
```

E2.1.82 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;
```

E2.1.83 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;
```

E2.1.84 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;
```

E2.1.85 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);
```

E2.1.86 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;
```

E2.1.87 FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) operand)
    assert N IN {32,64};
    return '0' : operand<N-2:0>;
```

E2.1.88 FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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<23:22> == '10' then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);
    return result;
```

E2.1.89 FPB_BreakpointMatch

```
// FPB_BreakpointMatch()
// =====================
// Generates a debug event based on FP Breakpoint Match

FPB_BreakpointMatch()
    i = GenerateDebugEventResponse();
```

E2.1.90 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;
```

E2.1.91 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;
```

E2.1.92 FPCompare

```
// FPCompare()
// ===========

(bit, bit, bit, bit) FPCompare(bits(N) op1, bits(N) op2, boolean quiet_nan_exc,
                               boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.93 FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN(integer N)
    assert N IN {16,32,64};
    if N == 16 then E = 5; elsif N == 32 then E = 8; else E = 11;
    constant integer F = N - E - 1;
    sign = '0';
    exp  = Ones(E);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

E2.1.94 FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.95 FPDoubleToHalf

```
// FPDoubleToHalf()
// ================
bits(16) FPDoubleToHalf(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<26> == '1' then // AH bit set
            result = FPZero(sign, 16);
        elsif fpscr_val<25> == '1' then // DN bit set
            result = FPDefaultNaN(16);
        else
            result = sign : '11111 1' : operand<50:42>;
        if fp_type == FPType_SNaN || fpscr_val<26> == '1' then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        if fpscr_val<26> == '1' then // AH bit set
            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 = FPRound(value, 16, fpscr_val);
    return result;
```

E2.1.96 FPDoubleToSingle

```
// FPDoubleToSingle()
// ==================

bits(32) FPDoubleToSingle(bits(64) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            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 = FPRound(value, 32, fpscr_val);
    return result;
```

E2.1.97 FPExc

```
// Floating point exceptions

enumeration FPExc {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                   FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

E2.1.98 FPHalfToDouble

```
// FPHalfToDouble()
// ================

bits(64) FPHalfToDouble(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<25> == '1' then // DN bit set
            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 = FPRound(value, 64, fpscr_val); // Rounding will be exact
    return result;
```

E2.1.99 FPHalfToSingle

```
// FPHalfToSingle()
// ================

bits(32) FPHalfToSingle(bits(16) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<25> == '1' then // DN bit set
            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 = FPRound(value, 32, fpscr_val); // Rounding will be exact
    return result;
```

E2.1.100 FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign, integer N)
    assert N IN {16,32,64};
    if N == 16 then E = 5; elsif N == 32 then E = 8; else E = 11;
    constant integer F = N - E - 1;
    exp  = Ones(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

E2.1.101 FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.102 FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign, integer N)
    assert N IN {16,32,64};
    if N == 16 then E = 5; elsif N == 32 then E = 8; else E = 11;
    constant integer F = N - E - 1;
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

E2.1.103 FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2)
    assert N IN {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);
```

E2.1.104 FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.105 FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2)
    assert N IN {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);
```

E2.1.106 FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.107 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 {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<23:22> == '10' then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);

    return result;
```

E2.1.108 FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) operand)
    assert N IN {32,64};
    return NOT(operand<N-1>) : operand<N-2:0>;
```

E2.1.109 FPProcessException

```
// FPProcessException()
// ====================
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

FPProcessException(FPExc exception, bits(32) 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;
```

E2.1.110 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, bits(32) fpscr_val)
    assert N IN {32,64};
    topfrac = if N == 32 then 22 else 51;
    result = operand;
    if fp_type == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpscr_val);
    if fpscr_val<25> == '1' then  // DefaultNaN requested
        result = FPDefaultNaN(N);
    return result;
```

E2.1.111 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 {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);
```

E2.1.112 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 {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);
```

E2.1.113 FPRound

```
// FPRound()
// =========
// The 'fpscr_val' argument supplies FPSCR control bits. Status information is
// updated directly in FPSCR where appropriate.

bits(N) FPRound(real value, integer N, bits(32) fpscr_val)
    assert N IN {16,32,64};
    assert value != 0.0;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then E = 5; elsif N == 32 then E = 8; else E = 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 fpscr_val<24> == '1' && N != 16 && 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 || fpscr_val<11> == '1') then
            FPProcessException(FPExc_Underflow, fpscr_val);

        // Round result according to rounding mode.
        case fpscr_val<23:22> of
            when '00'  // 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 '01'  // Round towards Plus Infinity
                round_up = (error != 0.0 && sign == '0');
                overflow_to_inf = (sign == '0');
            when '10'  // Round towards Minus Infinity
                round_up = (error != 0.0 && sign == '1');
                overflow_to_inf = (sign == '1');
            when '11'  // 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<26> == '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;
```

E2.1.114 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 {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;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, N, FPSCR);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, FPSCR);

    return result;
```

E2.1.115 FPSingleToDouble

```
// FPSingleToDouble()
// ==================

bits(64) FPSingleToDouble(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(64);
        else
            result = sign : '11111111111 1' : operand<21:0> : Zeros(29);
        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 = FPRound(value, 64, fpscr_val);  // Rounding will be exact
    return result;
```

E2.1.116 FPSingleToHalf

```
// FPSingleToHalf()
// ================

bits(16) FPSingleToHalf(bits(32) operand, boolean fpscr_controlled)
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    (fp_type,sign,value) = FPUnpack(operand, fpscr_val);
    if fp_type == FPType_SNaN || fp_type == FPType_QNaN then
        if fpscr_val<26> == '1' then     // AH bit set
            result = FPZero(sign, 16);
        elsif fpscr_val<25> == '1' then  // DN bit set
            result = FPDefaultNaN(16);
        else
            result = sign : '11111 1' : operand<21:13>;
        if fp_type == FPType_SNaN || fpscr_val<26> == '1' then
            FPProcessException(FPExc_InvalidOp, fpscr_val);
    elsif fp_type == FPType_Infinity then
        if fpscr_val<26> == '1' then // AH bit set
            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 = FPRound(value, 16, fpscr_val);
    return result;
```

E2.1.117 FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) operand)
    assert N IN {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;
```

E2.1.118 FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, boolean fpscr_controlled)
    assert N IN {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<23:22> == '10' then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, N, fpscr_val);
    return result;
```

E2.1.119 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 {32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_towards_zero then fpscr_val<23:22> = '11';
    (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<23:22> of
        when '00'  // Round to Nearest (rounding to even if exactly halfway)
            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 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;
```

E2.1.120 FPToFixedDirected

```
// FPToFixedDirected()
// ===================

bits(M) FPToFixedDirected(bits(N) op, integer fbits, boolean unsigned,
                          bits(2) round_mode, boolean fpscr_controlled)
    assert N IN {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;
```

E2.1.121 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};
```

E2.1.122 FPUnpack

```
// FPUnpack()
// ==========
//
// 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) FPUnpack(bits(N) fpval, bits(32) 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
            if IsZero(frac16) 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<26> == '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<24> == '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<24> == '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);
```

E2.1.123 FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign, integer N)
    assert N IN {16,32,64};
    if N == 16 then E = 5; elsif N == 32 then E = 8; else E = 11;
    constant integer F = N - E - 1;
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

E2.1.124 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;
```

E2.1.125 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, TRUE, TRUE);
        else
            UFSR.INVSTATE = '1';
            excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
        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, TRUE, TRUE);
            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, TRUE, TRUE);
            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, TRUE, TRUE);
        HandleException(excInfo);
    return (instr, isT16);
```

E2.1.126 FindPriv

```
// FindPriv()
// ==========

boolean FindPriv()
    return CurrentModeIsPrivileged();
```

E2.1.127 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 {32,64};
    fpscr_val = if fpscr_controlled then FPSCR else StandardFPSCRValue();
    if round_to_nearest then fpscr_val<23:22> = '00';
    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;
```

E2.1.128 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.
        _R[spName]     = framePtr + 8;
        IPSR.Exception = newPSR.Exception;
        CONTROL_S.SFPA   = newPSR.SFPA;
        // IT/ICI bits cleared to prevent Non-secure code interfering with
        // Secure execution
        if HaveMainExt() then
            ITSTATE = Zeros(8);
        // 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;
```

E2.1.129 GenerateCoprocessorException

```
// GenerateCoprocessorException()
// ==============================

GenerateCoprocessorException()
    UFSR.UNDEFINSTR = '1';
    excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
    HandleException(excInfo);
```

E2.1.130 GenerateDebugEventResponse

```
// GenerateDebugEventResponse()
// ============================
// Generate a debug event response based on the PE configuration.

boolean GenerateDebugEventResponse()
    if CanHaltOnEvent(IsSecure()) then
        DFSR.BKPT = '1';
        DHCSR.C_HALT = '1';
        return TRUE;
    elsif HaveMainExt() && CanPendMonitorOnEvent(IsSecure(), TRUE) then
        DFSR.BKPT = '1';
        DEMCR.MON_PEND = '1';
        excInfo = CreateException(DebugMonitor, FALSE, boolean UNKNOWN);
        HandleException(excInfo);
        return TRUE;
    else
        return FALSE;
```

E2.1.131 GenerateIntegerZeroDivide

```
// GenerateIntegerZeroDivide()
// ===========================

GenerateIntegerZeroDivide()
    UFSR.DIVBYZERO = '1';
    excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
    HandleException(excInfo);
```

E2.1.132 HaltingDebugAllowed

```
// HaltingDebugAllowed()
// =====================

boolean HaltingDebugAllowed()
    return ExternalInvasiveDebugEnabled() || DHCSR.S_HALT == '1';
```

E2.1.133 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) &&
               (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();
```

E2.1.134 HaveDSPExt

```
// HaveDSPExt()
// ===========
// Check whether DSP Extension is implemented.

boolean HaveDSPExt();
```

E2.1.135 HaveDWT

```
// HaveDWT()
// =========
// Check whether Data Watchpoint and Trace unit is implemented.

boolean HaveDWT();
```

E2.1.136 HaveDebugMonitor

```
// HaveDebugMonitor()
//===================

boolean HaveDebugMonitor()
    return HaveMainExt();
```

E2.1.137 HaveFPB

```
// HaveFPB()
// =========
// Check whether Flash Patch and Breakpoint unit is implemented.

boolean HaveFPB();
```

E2.1.138 HaveFPExt

```
// HaveFPExt()
// ===========
// Check whether Floating Point Extension is implemented.

boolean HaveFPExt();
```

E2.1.139 HaveHaltingDebug

```
// HaveHaltingDebug()
// ==================
// Check whether Halting debug implemented.

boolean HaveHaltingDebug();
```

E2.1.140 HaveITM

```
// HaveITM()
// =========
// Check whether Instrumentation Trace Macrocell is implemented.

boolean HaveITM();
```

E2.1.141 HaveMainExt

```
// HaveMainExt()
// =============
// Check whether Main Extension is implemented.

boolean HaveMainExt();
```

E2.1.142 HaveSPFPOnly

```
// HaveSPFPOnly()
// ===========
// Check whether Floating Point Extension only implementes single-precision.

boolean HaveSPFPOnly();
```

E2.1.143 HaveSecurityExt

```
// HaveSecurityExt()
// =================
// Check whether the implementation have Security Extensions.

boolean HaveSecurityExt();
```

E2.1.144 HaveSysTick

```
// HaveSysTick()
// =============
// Returns the number of SysTick instances (0, 1 or 2).

integer HaveSysTick();
```

E2.1.145 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;
```

E2.1.146 Hint_Debug

```
// Hint_Debug
// ==========
// Generate a hint to the debug system.

Hint_Debug(bits(4) option);
```

E2.1.147 Hint_PreloadData

```
// Hint_PreloadData
// ================
// Performs a preload data hint

Hint_PreloadData(bits(32) address);
```

E2.1.148 Hint_PreloadDataForWrite

```
// Hint_PreloadDataForWrite
// ========================
// Performs a preload data hint for write.

Hint_PreloadDataForWrite(bits(32) address);
```

E2.1.149 Hint_PreloadInstr

```
// Hint_PreloadInstr
// =================
// Performs a preload instructions hint

Hint_PreloadInstr(bits(32) address);
```

E2.1.150 Hint_Yield

```
// Hint_Yield
// ==========
// Performs a Yield hint

Hint_Yield();
```

E2.1.151 IDAUCheck

```
// IDAUCheck
// =========
// Query IDAU(Implementation Defined Attribution Unit) for attribution information

(boolean, boolean, boolean, bits(8), boolean) IDAUCheck(bits(32) address);
```

E2.1.152 ITAdvance

```
// ITAdvance()
// ===========

ITAdvance()
    if ITSTATE<2:0> == '000' then
        ITSTATE = '00000000';
    else
        ITSTATE<4:0> = LSL(ITSTATE<4:0>, 1);
```

E2.1.153 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;
```

E2.1.154 ITSTATEType

```
// If-Then execution state bits for the T32 IT instruction.

type ITSTATEType = bits(8);
```

E2.1.155 InITBlock

```
// InITBlock()
// ===========

boolean InITBlock()
    return (ITSTATE<3:0> != '0000');
```

E2.1.156 InstructionAdvance

```
// InstructionAdvance()
// ====================

InstructionAdvance(boolean instExecOk)
    // 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;
    EXC_RETURN_Type excReturn = NextInstrAddr();
    if _PendingReturnOperation then
        _PendingReturnOperation = FALSE;
        (excInfo, excReturn) = 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 occured on exception return, or whether it
                // occured 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 occured 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 HaveFPExt() then
                        for n = 0 to 31
                            S[n] = bits(32) UNKNOWN;
                    FPSCR = bits(32) UNKNOWN;
                    // 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 HaveFPExt() 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 calls to ThisInstrAddr(), NextInstrAddr(),
    // ThisInstrITState(), NextInstrITState() represent the context the
    // instruction was executed in. IE so the correct context is pushed to the
    // stack
    (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.
            nextExcReturn = if excInfo.inExcTaken then LR else excReturn;
            excInfo       = TailChain(exception, excIsSecure, nextExcReturn);
        else
            excInfo = ExceptionEntry(exception, excIsSecure, instExecOk);
        // Handle any derived faults that have occurred
        if excInfo.fault != NoFault then
            DerivedLateArrival(pePriority, peException, peIsSecure, excInfo,
                               exception, excIsSecure);

    // If the PC has moved away from the lockup address (eg because an NMI
    // has been taken) leave the lockup state.
    if DHCSR.S_LOCKUP == '1' && NextInstrAddr() != 0xEFFFFFFE<31:0> then
        DHCSR.S_LOCKUP = '0';
    // Only advance the PC and ITSTATE if not locked up.
    if DHCSR.S_LOCKUP != '1' then
        // Commit PC and ITSTATE changes ready for the next instruction.
        _R[RName_PC] = NextInstrAddr();
        _PCChanged   = FALSE;
        if HaveMainExt() then
            EPSR.IT         = NextInstrITState();
            _ITStateChanged = FALSE;
```

E2.1.157 InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// ===================================
// Perform an instruction synchronization barrier operation

InstructionSynchronizationBarrier(bits(4) option);
```

E2.1.158 Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

E2.1.159 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';
```

E2.1.160 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 = (CurrentMode() == PEMode_Handler) || (if isSecure then
                        CONTROL_S.nPRIV == '0' else CONTROL_NS.nPRIV == '0');
    (-, 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);
```

E2.1.161 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;
```

E2.1.162 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);
```

E2.1.163 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<cp*2> == '1' then
        enabled = FALSE;
        // Check SUS bit to determine the target state of any fault
        forceToSecure = CPPWR<(cp*2)+1> == '1';

    return (enabled, secure || forceToSecure);

(boolean, boolean) IsCPEnabled(integer cp)
    return IsCPEnabled(cp, CurrentModeIsPrivileged(), IsSecure());
```

E2.1.164 IsCPInstruction

```
// IsCPInstruction()
// =================

(boolean, integer) IsCPInstruction(bits(32) instr)
    isCp  = instr IN { '111x1110xxxxxxxxxxxxxxxxxxxxxxxx',
                       '111x110xxxxxxxxxxxxxxxxxxxxxxxxx' };
    cpNum = if isCp then UInt(instr<11:8>) else integer UNKNOWN;
    // CP 11 instructions are treated as CP10
    if cpNum == 11 then
        cpNum = 10;
    return (isCp, cpNum);
```

E2.1.165 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;
        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;

    // 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
```

E2.1.166 IsDWTEnabled

```
// IsDWTEnabled()
// ==============
// Check whether DWT is enabled.

boolean IsDWTEnabled()
   return HaveDWT() && DEMCR.TRCENA == '1' && NoninvasiveDebugAllowed();
```

E2.1.167 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;
```

E2.1.168 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);
```

E2.1.169 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);
```

E2.1.170 IsIrqValid

```
// IsIrqValid()
// ============
// Check whether given exception number denotes a valid external interrupt
// implemented by PE.

boolean IsIrqValid(integer e);
```

E2.1.171 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;
```

E2.1.172 IsSecure

```
// IsSecure()
// ==========

boolean IsSecure()
    return HaveSecurityExt() && CurrentState == SecurityState_Secure;
```

E2.1.173 LR

```
// LR
// ==

// Non-assignment form
bits(32) LR
    return R[14];

// Assignment form

LR = bits(32) value
    R[14] = value;
```

E2.1.174 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;
```

E2.1.175 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);
```

E2.1.176 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;
```

E2.1.177 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);
```

E2.1.178 LastInITBlock

```
// LastInITBlock()
// ===============

boolean LastInITBlock()
    return (ITSTATE<3:0> == '1000');
```

E2.1.179 LoadWritePC

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

LoadWritePC(bits(32) address, integer baseReg, bits(32) baseRegVal, boolean baseRegUpdate,
            boolean spLimCheck)

    if baseRegUpdate then
        regName    = LookUpRName(baseReg);
        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 = BXWritePC(address, FALSE);

    if baseRegUpdate && excInfo.fault != NoFault then
        // Restore the previous base reg value, SP limit checking is not performed
        _R[regName] = oldBaseVal;

    HandleException(excInfo);
```

E2.1.180 Lockup

```
// Lockup()
// ========

Lockup(boolean termInst)
    DHCSR.S_LOCKUP = '1';
    // Branch to the lockup address.
    BranchToAndCommit(0xEFFFFFFE<31:0>);
    if termInst then
        EndOfInstruction();
```

E2.1.181 LookUpRName

```
// LookUpRName()
// =============

RName LookUpRName(integer n)
    assert n >= 0 && n <= 15;
    case n of
        when 0  result = RName0;
        when 1  result = RName1;
        when 2  result = RName2;
        when 3  result = RName3;
        when 4  result = RName4;
        when 5  result = RName5;
        when 6  result = RName6;
        when 7  result = RName7;
        when 8  result = RName8;
        when 9  result = RName9;
        when 10 result = RName10;
        when 11 result = RName11;
        when 12 result = RName12;
        when 13 result = LookUpSP();
        when 14 result = RName_LR;
        when 15 result = RName_PC;
    return result;
```

E2.1.182 LookUpSP

```
// LookUpSP()
// ==========

RName LookUpSP()
    return LookUpSP_with_security_mode(IsSecure(), CurrentMode());
```

E2.1.183 LookUpSPLim

```
// LookUpSPLim()
// =============

(bits(32), boolean) LookUpSPLim(RName spreg)
    case spreg of
        when RNameSP_Main_Secure    limit = MSPLIM_S.LIMIT:'000';
        when RNameSP_Process_Secure limit = PSPLIM_S.LIMIT:'000';
        when RNameSP_Main_NonSecure
            limit = if HaveMainExt() then MSPLIM_NS.LIMIT:'000' else Zeros(32);
        when RNameSP_Process_NonSecure
            limit = if HaveMainExt() then PSPLIM_NS.LIMIT:'000' else Zeros(32);
        otherwise
            assert (FALSE);

    // 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.
    secure = ((spreg == RNameSP_Main_Secure) ||
              (spreg == RNameSP_Process_Secure));
    assert (!secure || HaveSecurityExt());
    if HaveMainExt() && IsReqExcPriNeg(secure) then
        ignLimit   = if secure then CCR_S.STKOFHFNMIGN else CCR_NS.STKOFHFNMIGN;
        applylimit = (ignLimit == '0');
    else
        applylimit = TRUE;

    return (limit, applylimit);
```

E2.1.184 LookUpSP_with_security_mode

```
// LookUpSP_with_security_mode()
// =============================

RName LookUpSP_with_security_mode(boolean isSecure, PEMode mode)
    RName 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 = RNameSP_Process_Secure;
        else
            sp = RNameSP_Process_NonSecure;
    else
        if isSecure then
            sp = RNameSP_Main_Secure;
        else
            sp = RNameSP_Main_NonSecure;
    return sp;
```

E2.1.185 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;
        if attrfield<1:0> != '00' then 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;
```

E2.1.186 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');

    // Get the MPU registers for the correct security domain
    if secure then
        mpu_ctrl = MPU_CTRL_S;
        mpu_type = MPU_TYPE_S;
        mair     = MPU_MAIR1_S:MPU_MAIR0_S;
    else
        mpu_ctrl = MPU_CTRL_NS;
        mpu_type = MPU_TYPE_NS;
        mair     = MPU_MAIR1_NS:MPU_MAIR0_NS;

    // Pre-compute if the execution priority is negative, as this can affect the
    // MPU permissions used. NOTE: If Non-secure FAULTMASK is set this is also
    // considered to be a negative priority for the purpose of the Non-secure
    // MPU permissions regardless of how Non-secure exceptions are prioritised
    // with respect to the Secure state.
    // 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
        negativePri = FPCCR_S.HFRDY == '0';
    else
        negativePri = IsReqExcPriNeg(secure);

    // 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 mpu_ctrl.ENABLE == '0' then
        if mpu_ctrl.HFNMIENA == '1' then UNPREDICTABLE;
        else hit = TRUE; // always use default map if MPU disabled
    elsif mpu_ctrl.HFNMIENA == '0' && negativePri then
        hit = TRUE;      // optionally use default for HardFault, NMI and FAULTMASK.
    else  // MPU is enabled so check each individual region
        if (mpu_ctrl.PRIVDEFENA == '1') && ispriv then
            hit = TRUE;  // optional default as background for Privileged accesses

        regionMatched = FALSE;
        for r = 0 to (UInt(mpu_type.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;
                        perms.xn          = rbar.XN;
                        perms.region      = r<7:0>;
                        perms.regionValid = TRUE;
                        hit               = TRUE;
                        sh                = rbar.SH;

                    // parsing MAIR0/1 Register fields
                    index        = UInt(rlar.AttrIndx);
                    attrfield    = mair<8*index+7:8*index>;
                    // decoding MAIR0/1 field and populating memory attributes
                    attributes   =  MAIRDecode(attrfield, sh);

    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);
```

E2.1.187 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);
```

E2.1.188 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);
```

E2.1.189 MaxExceptionNum

```
// MaxExceptionNum()
// =================
// Returns the maximum exception number supported

integer MaxExceptionNum()
    if HaveMainExt() then
        return 511;
    else
        return 47;
```

E2.1.190 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;
```

E2.1.191 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);
```

E2.1.192 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
            UFSR.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
                if acctype == AccType_STACK then
                    BFSR.UNSTKERR = '1';
                elsif acctype IN {AccType_NORMAL, AccType_ORDERED} then
                    BFAR.ADDRESS   = address;
                    BFSR.BFARVALID = '1';
                    BFSR.PRECISERR = '1';

            // Generate BusFault exception if it cannot be ignored.
            if !IsReqExcPriNeg(secure) || (CCR.BFHFNMIGN == '0') then
                // Create the exception. NOTE: If Main Extension is not implemented
                // the fault always escalates to a HardFault
                excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
        // PPB (0xE0000000 to 0xE0100000) is always little endian
        elsif AIRCR.ENDIANNESS == '1' && UInt(address<31:20>) != 0xE00 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
            UFSR.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
        // PPB (0xE0000000 to 0xE0100000) is always little endian
        if AIRCR.ENDIANNESS == '1' && UInt(address<31:20>) != 0xE00 then
            value = BigEndianReverse(value, size);

        if _Mem(memaddrdesc, size, value) then
            // Synchronous BusFault occurred. NOTE: async BusFaults are handled
            // in RaiseAsyncBusFault()

            // Check whether the execution priority is negative.
            // 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
                negativePri = FPCCR_S.HFRDY == '0';
            else
                negativePri = IsReqExcPriNeg(secure);

            if HaveMainExt() then
                if acctype == AccType_STACK then
                    BFSR.STKERR = '1';
                elsif acctype == AccType_LAZYFP then
                    BFSR.LSPERR = '1';
                elsif acctype IN {AccType_NORMAL, AccType_ORDERED} then
                    BFAR.ADDRESS   = address;
                    BFSR.BFARVALID = '1';
                    BFSR.PRECISERR = '1';

            // Generate BusFault exception if it cannot be ignored.
            if !negativePri || (CCR.BFHFNMIGN == '0') then
                // Create the exception. NOTE: If Main Extension is not implemented
                // the fault always escalates to a HardFault
                excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
    return excInfo;
```

E2.1.193 MemI

```
// MemI()
// ======

bits(16) MemI[bits(32) address]
    // Check permissions and get attributes
    // NOTE: The privilige flag passed to ValidateAddress may be overriden if
    //       the security of the memory is different from the current security
    //       state, eg when performing a Non-secure to Secure function call.
    (excInfo, memaddrdesc) = ValidateAddress(address, AccType_IFETCH, FindPriv(),
                                             IsSecure(), FALSE, TRUE);
    if excInfo.fault == NoFault then
        (error, value) = _Mem(memaddrdesc, 2);
        if error then
            value = bits(16) UNKNOWN;
            BFSR.IBUSERR = '1';
            // Create the exception. NOTE: If Main Extension is not implemented the fault
            // always escalates to a HardFault
            excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN);
    HandleException(excInfo);
    if IsDWTEnabled() then DWT_InstructionMatch(address);
    return value;
```

E2.1.194 MemO

```
// MemO[] - non-assignment form
// ============================

bits(8*size) MemO[bits(32) address, integer size]
    (excInfo, value) = MemA_with_priv_security(address, size, AccType_ORDERED,
                                               FindPriv(), IsSecure(), TRUE);
    HandleException(excInfo);
    return value;


// MemO[] - assignment form
// ========================

MemO[bits(32) address, integer size] = bits(8*size) value
    excInfo = MemA_with_priv_security(address, size, AccType_ORDERED, FindPriv(),
                                      IsSecure(), TRUE, value);
    HandleException(excInfo);
```

E2.1.195 MemType

```
// Types of memory

enumeration MemType {MemType_Normal, MemType_Device};
```

E2.1.196 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;
```

E2.1.197 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;
```

E2.1.198 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, FALSE, boolean UNKNOWN);
        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];
        // PPB (0xE0000000 to 0xE0100000) is always little endian
        if AIRCR.ENDIANNESS == '1' && UInt(address<31:20>) != 0xE00 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, FALSE, boolean UNKNOWN);
        HandleException(excInfo);
    else // if unaligned access
        // PPB (0xE0000000 to 0xE0100000) is always little endian
        if AIRCR.ENDIANNESS == '1' && UInt(address<31:20>) != 0xE00 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;
```

E2.1.199 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
)
```

E2.1.200 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);

        if aPri < bPri then
            exc  = a;
            pend = b;
        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;
```

E2.1.201 NextInstrAddr

```
// NextInstrAddr()
// ===============

bits(32) NextInstrAddr()
    if _PCChanged then
        return _NextInstrAddr;
    else
        return ThisInstrAddr() + ThisInstrLength();
```

E2.1.202 NextInstrITState

```
// NextInstrITState()
// ==================

ITSTATEType NextInstrITState()
    if HaveMainExt() then
        // If the IT state has been directly modified return that value as the
        // next state, otherwise advance the IT state normally.
        if _ITStateChanged then
            nextState = _NextInstrITState;
        else
            nextState = ThisInstrITState();
            if nextState<2:0> == '000' then
                nextState = '00000000';
            else
                nextState<4:0> = LSL(nextState<4:0>, 1);
    else
        nextState = Zeros(8);
    return nextState;
```

E2.1.203 NoninvasiveDebugAllowed

```
// NoninvasiveDebugAllowed()
// =========================

boolean NoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugEnabled() || HaltingDebugAllowed();
```

E2.1.204 PC

```
// PC - non-assignment form
// ========================
bits(32) PC
    return R[15];
```

E2.1.205 PEMode

```
// The PE execution modes.

enumeration PEMode {PEMode_Thread, PEMode_Handler};
```

E2.1.206 PendReturnOperation

```
// PendReturnOperation()
// =====================

PendReturnOperation(bits(32) returnValue)
    _NextInstrAddr         = returnValue;
    _PCChanged             = TRUE;
    _PendingReturnOperation = TRUE;
    return;
```

E2.1.207 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();
```

E2.1.208 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
)
```

E2.1.209 PopStack

```
// PopStack()
// ==========

ExcInfo PopStack(EXC_RETURN_Type excReturn)
    // 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 HaveFPExt() 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, TRUE, TRUE);

        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 + 0x28;

    // 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);
    BranchToAndCommit(pc);

    // Check the XPSR value that's 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, FALSE, boolean UNKNOWN);
    // 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 HaveFPExt() 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, TRUE, TRUE);
            // 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 unstack become
                // UNKNOWN in implementations that don't include the
                // Security Extensions.
                if exc.fault == NoFault then
                    for i = 0 to 15
                        if exc.fault == NoFault then
                            offset      = 0x20+(4*i);
                            (exc, S[i]) = Stack(frameptr, offset, spName, mode);
                    if exc.fault == NoFault then
                        (exc, FPSCR) = Stack(frameptr, 0x60, spName, mode);
                    if toSecure && FPCCR_S.TS == '1' then
                        for i = 0 to 15
                            if exc.fault == NoFault then
                                offset         = 0x68+(4*i);
                                (exc, S[i+16]) = Stack(frameptr, offset, spName, mode);
                        if exc.fault != NoFault then
                            for i = 16 to 31
                                S[i] = if HaveSecurityExt() then Zeros(32) else bits(32) UNKNOWN;
                    if exc.fault != NoFault then
                        for i = 0 to 15
                            S[i] = if HaveSecurityExt() then Zeros(32) else bits(32) UNKNOWN;
                        FPSCR    = if HaveSecurityExt() then Zeros(32) else bits(32) UNKNOWN;

    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
else
    APSR<31:28>      = psr<31:28>;            // Load valid APSR bits from memory
return exc;
```

E2.1.210 PreserveFPState

```
// PreserveFPState()
// =================

PreserveFPState()
    // 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'.

    isSecure = FPCCR_S.S == '1';
    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
    if !splimviol then

        // 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 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 a fault was raised handle it now. This function may call
    // EndOfInstruction(), as a result any code after this call may not execute.
    if excInfo.fault != NoFault then
        TakePreserveFPException(excInfo);

    // If the stores are interrupted, the register content and LSPACT remain unchanged.

    // If exception with sufficient priority to pre-empt current instruction execution
    // is raised during FP state preserve, then TakePreserveFPException() will terminate
    // the current  instruction by calling EndOfInstruction().
    // If the exception results in a lockup state, then TakePreserveFPException() will
    // enter the lockup state by calling Lockup().
    // In both above cases where execution of current instruction is not completed, either
    // by taking exception straight away or by entering lockup state, below code is not
    // executed and LSPACT is not cleared.
    // In case of NoFault or, on successful return from TakePreserveFPException(), the current
    // instruction execution continues and FPCCR.LSPACT will be cleared.

    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
    if isSecure && FPCCR_S.TS == '1' then
        for i = 0 to 31
            S[i] = Zeros(32);
        FPSCR = Zeros(32);
    else
        for i = 0 to 15
            S[i] = bits(32) UNKNOWN;
        FPSCR = bits(32) UNKNOWN;

    return;
```

E2.1.211 ProcessorID

```
// ProcessorID
// ===========
// Returns an integer that uniquely identifies the executing PE in the system.

integer ProcessorID();
```

E2.1.212 PushCalleeStack

```
// PushCalleeStack()
// =================

ExcInfo PushCalleeStack(boolean doTailChain)
    // allocate space of the correct stack. NOTE: If we are tail chaining we
    // look at LR 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 LR<3> == '0' then
            mode   = PEMode_Handler;
            spName = RNameSP_Main_Secure;
        else
            mode   = PEMode_Thread;
            spName = if LR<2> == '1' then RNameSP_Process_Secure else RNameSP_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 HaveFPExt() then 0xFEFA125A<31:1> : LR<4> else 0xFEFA125B<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, frameptr);
    return MergeExcInfo(exc, spExc);
```

E2.1.213 PushStack

```
// PushStack()
// ===========

ExcInfo PushStack(boolean secureException, boolean instExecOk)
    integer framesize;
    if HaveFPExt() && CONTROL.FPCA == '1' && (IsSecure() || NSACR.CP10 == '1') then
        if IsSecure() && FPCCR_S.TS == '1' then
            framesize = 0xA8;
        else
            framesize = 0x68;
    else
        framesize = 0x20;

    /* allocate space on the correct stack */
    bits(1) frameptralign;
    frameptralign = SP<2>;
    frameptr      = (SP - framesize) AND NOT(ZeroExtend('100',32));
    spName        = LookUpSP();

    /* only the stack locations, not the store order, are architected */
    (retaddr, itstate) = ReturnState(instExecOk);
    RETPSR_Type retpsr = XPSR<31:0>;
    retpsr.IT          = itstate; // 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();
    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, LR);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x18, spName, mode, retaddr);
    if exc.fault == NoFault then exc = Stack(frameptr, 0x1C, spName, mode, retpsr);

    if HaveFPExt() && CONTROL.FPCA == '1' then
        newExc = DefaultExcInfo();
        // LSPACT should not be active at the same time as CONTROL.FPCA. 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 HaveSecurityExt() && lspact == '1' then
            SFSR.LSERR = '1';
            newExc     = CreateException(SecureFault, TRUE, TRUE);
        elsif !IsSecure() && NSACR.CP10 == '0' then
            UFSR_S.NOCP = '1';
            newExc     = CreateException(UsageFault, TRUE, TRUE);
        else
            if 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, 0x20+(4*i), spName, mode, S[i]);
                    if exc.fault == NoFault then
                        exc = Stack(frameptr, 0x60, spName, mode, FPSCR);
                    if framesize == 0xA8 then
                        for i = 0 to 15
                            if exc.fault == NoFault then
                                exc = Stack(frameptr, 0x68+(4*i), spName, mode, S[i+16]);
                (cpEnabled, -) = IsCPEnabled(10);
                if cpEnabled then
                    if framesize == 0xA8 then
                        for i = 0 to 31
                            S[i] = Zeros(32);
                        FPSCR = Zeros(32);
                    else
                        for i = 0 to 15
                            S[i] = bits(32) UNKNOWN;
                        FPSCR    = bits(32) UNKNOWN;
            else
                UpdateFPCCR(frameptr + 0x20, 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, frameptr);
    exc   = MergeExcInfo(exc, spExc);

    bit isSecure = if IsSecure() then '1' else '0';
    bit isThread = if mode == PEMode_Thread then '1' else '0';
    // Some excReturn bits (eg ES, SPSEL) are set by ExceptionTaken
    if HaveFPExt() then
        LR = Ones(25):isSecure:'1':NOT(CONTROL.FPCA):isThread:'000';
    else
        LR = Ones(25):isSecure:'11':isThread:'000';
    return exc;
```

E2.1.214 R

```
// R[]
// ===

// Non-assignment form

bits(32) R[integer n]
    assert n >= 0 && n <= 15;
    bits(32) result;
    case n of
        when 0   result = _R[RName0];
        when 1   result = _R[RName1];
        when 2   result = _R[RName2];
        when 3   result = _R[RName3];
        when 4   result = _R[RName4];
        when 5   result = _R[RName5];
        when 6   result = _R[RName6];
        when 7   result = _R[RName7];
        when 8   result = _R[RName8];
        when 9   result = _R[RName9];
        when 10  result = _R[RName10];
        when 11  result = _R[RName11];
        when 12  result = _R[RName12];
        when 13  result = _R[LookUpSP()]<31:2>:'00';
        when 14  result = _R[RName_LR];
        when 15  result = _R[RName_PC] + 4;
    return result;

// Assignment form

R[integer n] = bits(32) value
    assert n >= 0 && n <= 14;
    RName regName;
    case n of
        when 0  _R[RName0]     = value;
        when 1  _R[RName1]     = value;
        when 2  _R[RName2]     = value;
        when 3  _R[RName3]     = value;
        when 4  _R[RName4]     = value;
        when 5  _R[RName5]     = value;
        when 6  _R[RName6]     = value;
        when 7  _R[RName7]     = value;
        when 8  _R[RName8]     = value;
        when 9  _R[RName9]     = value;
        when 10 _R[RName10]    = value;
        when 11 _R[RName11]    = value;
        when 12 _R[RName12]    = value;
        when 13
            // It is IMPLEMENTATION DEFINED whether stack pointer limit checking
            // is performed for instructions that were previously UNPREDICTABLE
            // when modifying the stack pointer.
            if boolean IMPLEMENTATION_DEFINED "SPLim check UNPRED instructions" then
                - = _SP(LookUpSP(), FALSE, value);
            else
                _R[LookUpSP()] = value<31:2>:'00';
        when 14 _R[RName_LR]   = value;
    return;
```

E2.1.215 RName

```
// The names of the core registers. SP is a Banked register.

enumeration RName {RName0, RName1, RName2, RName3, RName4, RName5, RName6,
                   RName7, RName8, RName9, RName10, RName11, RName12,
                   RNameSP_Main_NonSecure, RNameSP_Process_NonSecure, RName_LR, RName_PC,
                   RNameSP_Main_Secure, RNameSP_Process_Secure};
```

E2.1.216 ROR

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

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

E2.1.217 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);
```

E2.1.218 RRX

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

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

E2.1.219 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);
```

E2.1.220 RSPCheck

```
// RSPCheck[] - assignment form
// ============================

RSPCheck[integer n] = bits(32) value
    if n == 13 then
        - = _SP(LookUpSP(), FALSE, value);
    else
        R[n] = value;
    return;
```

E2.1.221 RaiseAsyncBusFault

```
// RaiseAsyncBusFault()
// ====================

RaiseAsyncBusFault()
    if HaveMainExt() then
        BFSR.IMPRECISERR = '1';

    excInfo = CreateException(BusFault, FALSE, boolean UNKNOWN, FALSE);
    HandleException(excInfo);
```

E2.1.222 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;
    return execPri;
```

E2.1.223 ResetSCSRegs

```
// ResetSCSRegs
// ============
// Sets all registers in the System Control Space (SCS) that have
// architecturally-defined reset values to those values

ResetSCSRegs();
```

E2.1.224 RestrictedNSPri

```
// RestrictedNSPri()
// =================
// The priority to which Non-secure exceptions are restricted if AIRCR.PRIS is set

integer RestrictedNSPri()
    return 0x80;
```

E2.1.225 ReturnState

```
// ReturnState()
// =============

(bits(32), ITSTATEType) ReturnState(boolean instExecOk)

    // 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 instExecOk then
        return (NextInstrAddr(), NextInstrITState());
    else
        return (ThisInstrAddr(), ThisInstrITState());
```

E2.1.226 S

```
// S[]
// ===

// Non-assignment form

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
        result = D[n DIV 2]<31:0>;
    else
        result = D[n DIV 2]<63:32>;
    return result;

// Assignment form

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    if (n MOD 2) == 0 then
        D[n DIV 2]<31:0> = value;
    else
        D[n DIV 2]<63:32> = value;
    return;
```

E2.1.227 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
)
```

E2.1.228 SCS_UpdateStatusRegs

```
// SCS_UpdateStatusRegs()
// ======================
// Update status registers in the System Control Space (SCS)

SCS_UpdateStatusRegs();
```

E2.1.229 SP

```
// SP
// ==

// Non-assignment form

bits(32) SP
    return R[13];

// Assignment form

SP = bits(32) value
    RSPCheck[13] = value;
```

E2.1.230 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;
```

E2.1.231 SP_Main_NonSecure

```
// SP_Main_NonSecure
// =================

// Non-assignment form

bits(32) SP_Main_NonSecure
    return _SP(RNameSP_Main_NonSecure);

// Assignment form

SP_Main_NonSecure = bits(32) value
    - = _SP(RNameSP_Main_NonSecure, FALSE, value);
```

E2.1.232 SP_Main_Secure

```
// SP_Main_Secure
// ==============

// Non-assignment form

bits(32) SP_Main_Secure
    return _SP(RNameSP_Main_Secure);

// Assignment form

SP_Main_Secure = bits(32) value
    - = _SP(RNameSP_Main_Secure, FALSE, value);
```

E2.1.233 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;
```

E2.1.234 SP_Process_NonSecure

```
// SP_Process_NonSecure
// ====================

// Non-assignment form

bits(32) SP_Process_NonSecure
    return _SP(RNameSP_Process_NonSecure);

// Assignment form

SP_Process_NonSecure = bits(32) value
    - = _SP(RNameSP_Process_NonSecure, FALSE, value);
```

E2.1.235 SP_Process_Secure

```
// SP_Process_Secure
// =================

// Non-assignment form

bits(32) SP_Process_Secure
    return _SP(RNameSP_Process_Secure);

// Assignment form

SP_Process_Secure = bits(32) value
    - = _SP(RNameSP_Process_Secure, FALSE, value);
```

E2.1.236 SRType

```
// Different types of shift and rotate operations
enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

E2.1.237 Sat

```
// Sat()
// =====

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

E2.1.238 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);
```

E2.1.239 SecureDebugMonitorAllowed

```
// SecureDebugMonitorAllowed()
// ===========================

boolean SecureDebugMonitorAllowed()
    if DAUTHCTRL.SPIDENSEL == '1' then
        return DAUTHCTRL.INTSPIDEN == '1';
    else
        return ExternalSecureSelfHostedDebugEnabled();
```

E2.1.240 SecureHaltingDebugAllowed

```
// SecureHaltingDebugAllowed()
// ===========================

boolean SecureHaltingDebugAllowed()
    if HaltingDebugAllowed() == FALSE then
        return FALSE;
    elsif DAUTHCTRL.SPIDENSEL == '1' then
        return DAUTHCTRL.INTSPIDEN == '1';
    else
        return ExternalSecureInvasiveDebugEnabled();
```

E2.1.241 SecureNoninvasiveDebugAllowed

```
// SecureNoninvasiveDebugAllowed()
// ===============================

boolean SecureNoninvasiveDebugAllowed()
    if !NoninvasiveDebugAllowed() then
        return FALSE;
    elsif DHCSR.S_SDE == '1' then
        return TRUE;
    elsif DAUTHCTRL.SPNIDENSEL == '1' then
        return DAUTHCTRL.INTSPNIDEN == '1';
    else
        return ExternalSecureNoninvasiveDebugEnabled();
```

E2.1.242 SecurityCheck

```
// SecurityCheck()
// ===============

SAttributes SecurityCheck(bits(32) address, boolean isinstrfetch, boolean isSecure)
    SAttributes result;
    addr = UInt(address);

    // Setup default attributes
    result.ns      = FALSE;
    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 <= 0xE0002FFF)) ||   // ITM, DWT, FPB
          ((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;
```

E2.1.243 SecurityState

```
// Type and definition of the current Security state of PE

enumeration SecurityState {SecurityState_NonSecure, SecurityState_Secure};
SecurityState CurrentState;
```

E2.1.244 SendEvent

```
// SendEvent
// =========
// Performs a send event by setting the Event Register of every PE in multiprocessor system

SendEvent();
```

E2.1.245 SerializeVFP

```
// SerializeVFP
// ============
// Ensures that any exceptional conditions in previous floating-point
// instructions have been detected

SerializeVFP();
```

E2.1.246 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';
```

E2.1.247 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) then
        DEMCR.MON_PEND = '1';
        DFSR.DWTTRAP = '1';
        return TRUE;

    else
        return FALSE;
```

E2.1.248 SetEventRegister

```
// SetEventRegister()
// ==================
// Set the Event Register of the current PE

SetEventRegister();
```

E2.1.249 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);
```

E2.1.250 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;
```

E2.1.251 SetMonStep

```
// SetMonStep()
// ============
// Check whether DebugMonitor priority is still sufficient for debug stepping after the
// execution of current instruction. If monitor step is enabled before the
// execution of the instruction and the priority remains sufficient after the execution
// of current instruction, then MON_PEND bit is set for current instruction.

SetMonStep(boolean mon_step_active)

    // Check whether Monitor Step is enabled at the start of the instruction
    if !mon_step_active then return;

    // if Monitor Step is enabled, check whether current instruction has cleared MON_STEP bit
    if DEMCR.MON_STEP == '0' then UNPREDICTABLE;

    // Check whether DebugMonitor priority remains greater-than the current priority, and if so,
    // set the MON_PEND bit.
    if ExceptionPriority(DebugMonitor, IsSecure(), TRUE) < ExecutionPriority() then
        DEMCR.MON_PEND = '1';
        DFSR.HALTED = '1';
    return;
```

E2.1.252 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';
```

E2.1.253 SetThisInstrDetails

```
// SetThisInstrDetails
// ===================
// Set the details of current instruction

SetThisInstrDetails(bits(32) opcode, integer len, bits(4) defaultCond);
```

E2.1.254 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;
```

E2.1.255 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);
```

E2.1.256 SignedSat

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

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

E2.1.257 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);
```

E2.1.258 SleepOnExit

```
// SleepOnExit()
// =============
// Optionally returns PE to a power-saving mode on return from the only
// active exception

SleepOnExit();
```

E2.1.259 Stack

```
// Stack
// =====

// Assignment form

ExcInfo Stack(bits(32) frameptr, integer offset, RName 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.
    (limit, applylimit) = LookUpSPLim(spreg);
    if !applylimit || (UInt(frameptr) >= UInt(limit)) then
        doAccess = TRUE;
    else
        doAccess = boolean IMPLEMENTATION_DEFINED "Push non-violating locations";

    address = frameptr + offset;
    if doAccess && (!applylimit || ((UInt(address) >= UInt(limit)))) then
        secure = ((spreg == RNameSP_Main_Secure)   ||
                  (spreg == RNameSP_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, RName spreg, PEMode mode)
    secure = ((spreg == RNameSP_Main_Secure)   ||
              (spreg == RNameSP_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);
```

E2.1.260 StandardFPSCRValue

```
// StandardFPSCRValue()
// ====================

bits(32) StandardFPSCRValue()
    return '00000' : FPSCR<26> : '11000000000000000000000000';
```

E2.1.261 SteppingDebug

```
// SteppingDebug()
// ===============
// At the start of each instruction execution, check for debug stepping.
// This function does not cover the scenario where the instruction being stepped raises another
// exception, or returns from an exception and enters/tailchains into another exception without
// executing the instruction in background code.

boolean SteppingDebug()
    // If Halting debug is allowed and C_STEP is set, set C_HALT for the next instruction.
    if CanHaltOnEvent(IsSecure()) && DHCSR.C_STEP == '1' then
        DHCSR.C_HALT = '1';
        DFSR.HALTED = '1';

    // If the current execution priority is below DebugMonitor and generating a DebugMonitor
    // exception is allowed, and MON_STEP is set, then return TRUE. Otherwise return FALSE.
    // This is used to determine whether to set MON_PEND for the next instruction if the
    // execution priority remains below DebugMonitor.
    mon_step_enabled = HaveDebugMonitor() && CanPendMonitorOnEvent(IsSecure(), FALSE);
    return (mon_step_enabled && DEMCR.MON_STEP == '1');
```

E2.1.262 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;
```

E2.1.263 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);
```

E2.1.264 TTResp

```
// TTResp()
// ========

bits(32) TTResp(bits(32) address, boolean alt, boolean forceunpriv)
    TT_RESP_Type resp = Zeros();

    // Only allow security checks if currently in Secure state
    if IsSecure() then
        sAttributes = SecurityCheck(address, FALSE, IsSecure());
        if sAttributes.srvalid then
            resp.SREGION = sAttributes.sregion;
            resp.SRVALID = '1';
        if sAttributes.irvalid then
            resp.IREGION = sAttributes.iregion;
            resp.IRVALID = '1';
        addrSecure = if sAttributes.ns  then '0' else '1';
        resp.S     = addrSecure;

    // MPU region information only available when privileged or when
    // inspecting the other MPU state.
    other_domain  = (alt != IsSecure());
    if CurrentModeIsPrivileged() || alt then
        (write, read, region, hit) = IsAccessible(address, forceunpriv, other_domain);
        if hit then
            resp.MREGION = region;
            resp.MRVALID = '1';
        resp.R  = read;
        resp.RW = write;
        if IsSecure() then
            resp.NSR  = read  AND NOT addrSecure;
            resp.NSRW = write AND NOT addrSecure;

    return resp;
```

E2.1.265 TailChain

```
// TailChain()
// ===========

ExcInfo TailChain(integer exceptionNumber, boolean excIsSecure, EXC_RETURN_Type excReturn)
    // Refresh LR with the excReturn value, ready for the next exception
    if !HaveFPExt() then
        excReturn.FType = '1';
    excReturn.PREFIX = Ones(8);
    LR = excReturn;

    return ExceptionTaken(exceptionNumber, TRUE, excIsSecure, FALSE);
```

E2.1.266 TakePreserveFPException

```
// TakePreserveFPException()
// ========================

TakePreserveFPException(ExcInfo excInfo)
    assert HaveFPExt();
    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;

    // 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.
    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);
        EndOfInstruction();
    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(TRUE); // Lockup at current priority, lock-up address = 0xEFFFFFFE
        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;
```

E2.1.267 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;

    ResetSCSRegs();                     // Catch-all function for System Control Space reset
    APSR           = bits(32) UNKNOWN;  // Flags UNPREDICTABLE from reset
    IPSR.Exception = Zeros(9);          // Exception number cleared at 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

    // Reset priority boosting
    PRIMASK_NS<0> = '0';                // priority mask cleared at reset
    if HaveSecurityExt() then
        PRIMASK_S<0> = '0';
    if HaveMainExt() then
        FAULTMASK_NS<0> = '0';          // Fault mask cleared at reset
        BASEPRI_NS<7:0> = Zeros(8);     // Base priority disabled at reset
        if HaveSecurityExt() then
            FAULTMASK_S<0> = '0';
            BASEPRI_S<7:0> = Zeros(8);

    // Initialize the Floating Point Extn
    if HaveFPExt() then
        CONTROL.FPCA    = '0';          // FP inactive
        FPDSCR_NS.AHP   = '0';
        FPDSCR_NS.DN    = '0';
        FPDSCR_NS.FZ    = '0';
        FPDSCR_NS.RMode = '00';
        FPCCR.LSPEN     = '1';
        FPCCR_NS.ASPEN  = '1';
        FPCCR_NS.LSPACT = '0';
        FPCAR_NS        = bits(32) UNKNOWN;
        if HaveSecurityExt() then
            CONTROL_S.SFPA   = '0';
            FPDSCR_S.AHP   = '0';
            FPDSCR_S.DN    = '0';
            FPDSCR_S.FZ    = '0';
            FPDSCR_S.RMode = '00';
            FPCCR.LSPENS   = '0';
            FPCCR_S.ASPEN  = '1';
            FPCCR_S.LSPACT = '0';
            FPCAR_S        = bits(32) UNKNOWN;
        for i = 0 to 31
            S[i] = bits(32) UNKNOWN;

    // Thread is privileged, current stack is Main
    CONTROL_NS.SPSEL    = '0';
    CONTROL_NS.nPRIV    = '0';
    if HaveSecurityExt() then
        CONTROL_S.SPSEL = '0';
        CONTROL_S.nPRIV = '0';

    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;
    // Stack limit registers. It is IMPLEMENTATION DEFINED how many bits of
    // these registers are writable. The following writes only affect the
    // bits that an implementation defines as writable
    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>;
    BranchToAndCommit(start<31:1>:'0');
```

E2.1.268 ThisInstr

```
// ThisInstr
// =========
// Returns a 32-bit value which contain the bitstring encoding of current instruction.
// In case of 16-bit instructions, the instruction is packed into the bottom 16-bits
// with upper 16-bits zeroed. In case of 32-bit instructions, the instruction is
// treated as two halfwords, with the first halfword of the instruction in the
// top 16-bits and second halfword in bottom 16-bits.

bits(32) ThisInstr();
```

E2.1.269 ThisInstrAddr

```
// ThisInstrAddr()
// ===============

bits(32) ThisInstrAddr()
    return _R[RName_PC];
```

E2.1.270 ThisInstrITState

```
// ThisInstrITState()
// ==================

ITSTATEType ThisInstrITState()
    if HaveMainExt() then
        value = EPSR.IT;
    else
        value = Zeros(8);
    return value;
```

E2.1.271 ThisInstrLength

```
// ThisInstrLength
// ===============
// Returns the length of the current instruction in bytes

integer ThisInstrLength();
```

E2.1.272 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()
    // If the PE is locked up then abort execution of this instruction. Set
    // the length of the current instruction to 0 so NextInstrAddr() reports the
    // correct lockup address.
    ok = DHCSR.S_LOCKUP != '1';
    if !ok then
        SetThisInstrDetails(Zeros(32), 0, Ones(4));
    else
        // Check for stepping debug for current instruction fetch.
        mon_step_active = SteppingDebug();
        UpdateSecureDebugEnable();
        pc = ThisInstrAddr();

        try
            // Not locked up, so attempt to fetch the instruction
            (instr, is16bit) = FetchInstr(pc);

            // Setup the details of the instruction. NOTE: The default condition
            // is based on the ITSTATE, however this is overridden in the decode
            // stage by instructions that have explicit condition codes.
            len         = if is16bit then 2 else 4;

            defaultCond = if ITSTATE<3:0> == 0 then 0xE<3:0> else ITSTATE<7:4>;
            SetThisInstrDetails(instr, len, defaultCond);

            // Checking for FPB Breakpoint on instructions
            if HaveFPB() && FPB_CheckBreakPoint(pc, len, TRUE, IsSecure()) then
                FPB_BreakpointMatch();

            // Finally try and execute the instruction
            DecodeExecute(instr, pc, is16bit);

            // Check for Monitor Step
            if HaveDebugMonitor() then SetMonStep(mon_step_active);

            // Check for DWT match
            if IsDWTEnabled() then DWT_InstructionMatch(pc);

        catch exn
            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 {'0000000000000000010111111xxxx0000',
                                    '111100111010111110000000xxxxxxxx'};
                if ConditionHolds(CurrentCond()) && !nopHint then
                    ok       = 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);
            when IsExceptionTaken(exn)
                ok = FALSE;
            // Do not catch UNPREDICTABLE or internal errors

    // If there is a reset pending do that, otherwise process the normal
    // instruction advance.
    try
        if ExceptionPending[Reset] != '00' then
            ExceptionPending[Reset] = '00';
            TakeReset();
        else
            // Call instruction advance for exception handling and PC/ITSTATE
            // advance.
            InstructionAdvance(ok);
    catch exn
        // Do not catch UNPREDICTABLE or internal errors
        when IsExceptionTaken(exn)
            // The correct architectural behavior for any exceptions is
            // performed inside TakeReset() and InstructionAdvance(). So no
            // additional actions are required in this catch block.
```

E2.1.273 UnsignedSat

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

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

E2.1.274 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);
```

E2.1.275 UpdateFPCCR

```
// UpdateFPCCR()
// =============

UpdateFPCCR(bits(32) frameptr, boolean applySpLim)
    assert(HaveFPExt());

    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.
    (limit, limitValid) = LookUpSPLim(LookUpSP());
    if applySpLim && limitValid && (UInt(frameptr) < UInt(limit)) 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';
    monpri = ExceptionPriority(DebugMonitor, DEMCR.SDME == '1', FALSE);
    if DEMCR.MON_EN == '1' && execPri > monpri then
        FPCCR_S.MONRDY = '1';
    else
        FPCCR_S.MONRDY = '0';
    return;
```

E2.1.276 UpdateSecureDebugEnable

```
// UpdateSecureDebugEnable()
// ========================
// Update DHCSR.S_SDE and DEMCR.SDME for each instruction

UpdateSecureDebugEnable()

    // DHCSR.S_SDE is frozen if the PE is in Debug state
    if DHCSR.S_HALT == '0' then
        DHCSR.S_SDE = (if SecureHaltingDebugAllowed() 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');
```

E2.1.277 VFPExcBarrier

```
// VFPExcBarrier
// =============
// Ensures that all floating-point exception processing has completed

VFPExcBarrier();
```

E2.1.278 VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8, integer N)
    assert N IN {32,64};
    if N == 32 then E = 8; else E = 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;
```

E2.1.279 VFPNegMul

```
// Different types of floating-point multiply and negate operations

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};
```

E2.1.280 VFPSmallRegisterBank

```
// VFPSmallRegisterBank
// ====================
// Returns TRUE if Floating Point implementation provides access only to
// 16 double-precision registers

boolean VFPSmallRegisterBank();
```

E2.1.281 ValidateAddress

```
// ValidateAddress()
// =================

(ExcInfo, AddressDescriptor) ValidateAddress(bits(32) address, AccType acctype,
                                             boolean ispriv, boolean secure,
                                             boolean iswrite, boolean aligned)
    AddressDescriptor result;
    Permissions       perms;
    ns               = boolean UNKNOWN;
    excInfo          = DefaultExcInfo();
    isInstrfetch     = acctype == AccType_IFETCH;

    // Security checking and MPU bank selection if Security Extensions are present.
    if HaveSecurityExt() then
        // Check SAU/IDAU for given address.
        sAttrib = SecurityCheck(address, isInstrfetch, secure);
        if isInstrfetch then
            ns        = sAttrib.ns;
            secureMpu = !sAttrib.ns;
            // Override the privilege flag supplied with the a value based on the
            // privilege associated with the current mode and the Security state
            // of the MPU being queried. This can be different from value this
            // function is called with, because CONTROL.nPRIV is banked between
            // the Security states.
            ispriv = CurrentModeIsPrivileged(secureMpu);
        else
            ns        = !secure || sAttrib.ns;
            secureMpu = secure;
    else
        ns        = TRUE;
        secureMpu = FALSE;

    // Getting memory attribute information from MPU. Note that NS information
    // in the memory attribute is set by  SAU/IDAU and is updated after getting
    // attribute values from MPU.
    (result.memattrs, perms) = MPUCheck(address, acctype, ispriv, secureMpu);
    // Updating NS information got from SAU/IDAU in memory attributes
    result.memattrs.NS       = ns;

    // Generate UNALIGNED UsageFault exception if access to Device memory is unaligned.
    if !aligned && result.memattrs.memtype == MemType_Device && perms.apValid == TRUE then
        UFSR.UNALIGNED = '1';
        excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);

    if excInfo.fault == NoFault && HaveSecurityExt() then
        // Check if there is a SAU/IDAU violation and, if so, update the fault informations
        raiseSecFault = FALSE;
        if isInstrfetch then
            if secure then
                if sAttrib.ns then
                    // Invalid exit from the Secure state
                    SFSR.INVTRAN   = '1';
                    raiseSecFault = TRUE;
            else
                if !sAttrib.ns && !sAttrib.nsc then
                    // Invalid entry to the Secure state
                    SFSR.INVEP    = '1';
                    raiseSecFault = TRUE;
        else
            if !secure && !sAttrib.ns then
                // Vector table faults don't generate SFAR/SFSR syndrome info. They are
                // reported via HFSR.VECTTBL which is not set here.
                if HaveMainExt() && acctype != AccType_VECTABLE then
                    if acctype == AccType_LAZYFP then
                        SFSR.LSPERR = '1';
                    else
                        SFSR.AUVIOL = '1';
                    SFSR.SFARVALID = '1';
                    SFAR           = address;
                // If Main Extension is not implemented the fault always escalates to a HardFault.
                raiseSecFault = TRUE;

        if raiseSecFault then
            excInfo = CreateException(SecureFault, TRUE, TRUE);

    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);
```

E2.1.282 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 !HaveFPExt() && excReturn.FType == '0' then
        UNPREDICTABLE;

    // Security specific validation
    targetDomainSecure = excReturn.ES == '1';
    if HaveSecurityExt() then
        // The state of the exception is considered to be Non-secure if
        // returning from the Non-secure state (returns from Secure exceptions
        // whilst in the Non-secure state are not permitted), or if
        // excReturn.ES == 0. This is to deal with Non-secure exceptions that
        // function chain to a Secure function when returning (that is, pass
        // excReturn in LR to a Secure function).
        excStateNonSecure = CurrentState == SecurityState_NonSecure || !targetDomainSecure;

        // Check DCRS bit not used in Non-secure state, also check that this
        // is not an attempt to retire a Secure exception from the Non-secure
        // state
        if excStateNonSecure && (excReturn.DCRS == '0' || targetDomainSecure) then
            // 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.
            if HaveMainExt() then
                SFSR.INVER = '1';

            // If exception return is invalide and is attempted from Non-secure state with
            // EXC_RETURN.ES set as '1', then ES should be treated as '0'
            if excStateNonSecure && targetDomainSecure then
                excReturn.ES = '0';

            targetDomainSecure = FALSE;
            error              = TRUE;
            exceptionNumber    = SecureFault;
    else
        excStateNonSecure = TRUE;

    // check returning from an inactive handler
    if !error then
        if !IsActiveForState(returningExceptionNumber, targetDomainSecure) then
            error = TRUE;
            if HaveMainExt() then
                UFSR.INVPC      = '1';
                exceptionNumber = UsageFault;
            else
                exceptionNumber = HardFault;

    if error then
        DeActivate(returningExceptionNumber, targetDomainSecure);
        if HaveSecurityExt() && targetDomainSecure 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, boolean UNKNOWN);
    else
        excInfo = DefaultExcInfo();
    return (excInfo, excReturn);
```

E2.1.283 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. They are therefore treated as HardFaults
    if exc.fault != NoFault then
        exc.isTerminal = TRUE;
        exc.fault      = HardFault;
        exc.isSecure   = exc.isSecure || AIRCR.BFHFNMINS == '0';
        HFSR.VECTTBL   = '1';
    return (exc, vector);
```

E2.1.284 WaitForEvent

```
// WaitForEvent
// ============
// Optionally suspends execution until a WFE wakeup event or reset occurs,
// or until some earlier time if the implementation chooses

WaitForEvent();
```

E2.1.285 WaitForInterrupt

```
// WaitForInterrupt
// ================
// Optionally suspends execution until a WFI wakeup event or reset occurs, or
// until some earlier time if the implementation chooses

WaitForInterrupt();
```

E2.1.286 _D

```
// The 32-bit extension register bank for the FP extension.

array bits(64) _D[0..15];
```

E2.1.287 _ITStateChanged

```
// Indicates a write to ITSTATE

boolean  _ITStateChanged;
```

E2.1.288 _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;
```

E2.1.289 _NextInstrAddr

```
// Address of  next instruction to be fetched in case of branch type operation

bits(32) _NextInstrAddr;
```

E2.1.290 _NextInstrITState

```
// Updated ITSTATE for next instruction

ITSTATEType _NextInstrITState;
```

E2.1.291 _PCChanged

```
// Indicates a change in instruction fetch address due to branch type operations

boolean  _PCChanged;
```

E2.1.292 _PendingReturnOperation

```
// Indicate any pending exception returns

boolean _PendingReturnOperation;
```

E2.1.293 _R

```
// The physical array of core registers.
// _R[RName_PC] is defined to be the address of the current instruction.
// The offset of 4 bytes is applied to it by the register access functions.

array bits(32) _R[RName];
```

E2.1.294 _SP

```
// _SP()
// =====

// Non-assignment form

bits(32) _SP(RName spreg)
    assert ( (spreg == RNameSP_Main_NonSecure)                           ||
            ((spreg == RNameSP_Main_Secure)    && HaveSecurityExt()) ||
             (spreg == RNameSP_Process_NonSecure)                        ||
            ((spreg == RNameSP_Process_Secure) && HaveSecurityExt()) );

    return _R[spreg]<31:2>:'00';

// Assignment form

ExcInfo _SP(RName spreg, boolean excEntry, bits(32) value)
    excInfo = DefaultExcInfo();
    (limit, applylimit) = LookUpSPLim(spreg);
    if applylimit && (UInt(value) < UInt(limit)) then
        // 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
            _R[spreg] = limit;

        // Raise the appropriate exception and syndrome information
        if HaveMainExt() then
            UFSR.STKOF = '1';
        // Create the exception. NOTE: If Main Extension is not implemented the fault always
        // escalates to HardFault.
        excInfo = CreateException(UsageFault, FALSE, boolean UNKNOWN);
        if !excEntry then
            HandleException(excInfo);
    else
        // Stack pointer only updated normally if limit not violated
        _R[spreg] = value<31:2>:'00';
    return excInfo;
```

E2.1.295 common

```
// 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;



// 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>;



// 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;



// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);



// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - 1 - HighestSetBit(x);



// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;



// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);



// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';



// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;



// 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;



// 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;



// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

bits(N) Ones()
    return Ones(N);



// 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);
```

```
// RoundDown()
// ===========

integer RoundDown(real x);



// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x > 0.0 then RoundDown(x) else RoundUp(x);



// RoundUp()
// =========

integer RoundUp(real x);



// 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);



// 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);



// 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:

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* on page F1-1363
- *Packet transmission by the trace sink* on page F1-1363

——— **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

Table F1-1 8-bit header encodings

[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Description
0	0	0	0	0	0	0	0	*Synchronization packet* on page F1-1385
0	1	1	1	0	0	0	0	*Overflow packet* on page F1-1383
0	≠ 0b000 && ≠ 0b111			0	0	0	0	*Local Timestamp 2 packet* on page F1-1382
1	0	0	1	0	1	0	0	*Global Timestamp 1 packet* on page F1-1375
1	0	1	1	0	1	0	0	*Global Timestamp 2 packet* on page F1-1377
1	1	x	x	0	0	0	0	*Local Timestamp 1 packet* on page F1-1380
x	x	x	x	1	x	0	0	*Extension packet* on page F1-1373
0	0	0	0	0	1	0	1	*Event Counter packet* on page F1-1371
0	1	x	x	0	1	0	1	*Data Trace Match packet* on page F1-1368
0	0	0	0	1	1	1	0	*Exception Trace packet* on page F1-1372
0	1	x	x	0	1	≠ 0b00		*Data Trace PC Value packet* on page F1-1369
0	1	x	x	1	1	≠ 0b00		*Data Trace Data Address packet* on page F1-1365
1	0	x	x	x	1	≠ 0b00		*Data Trace Data Value packet* on page F1-1366
x	x	x	x	x	0	≠ 0b00		*Instrumentation packet* on page F1-1379
0	0	0	1	0	1	x	1	*Periodic PC Sample packet* on page F1-1384

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. Figure F1-1 on page F1-1364 shows this convention, and how it relates to data transmission for a packet with a header byte and two payload bytes.

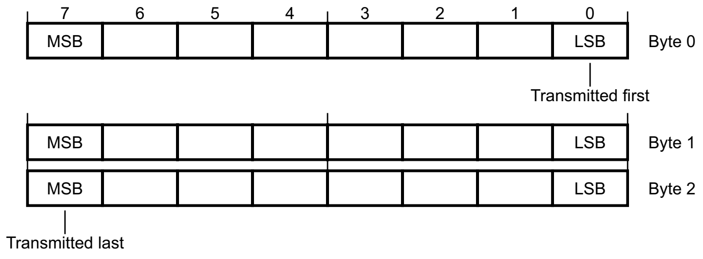


Figure F1-1 Convention for packet descriptions

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

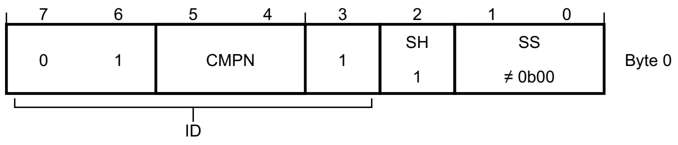
F1.2.1 Data Trace Data Address packet

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.

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]

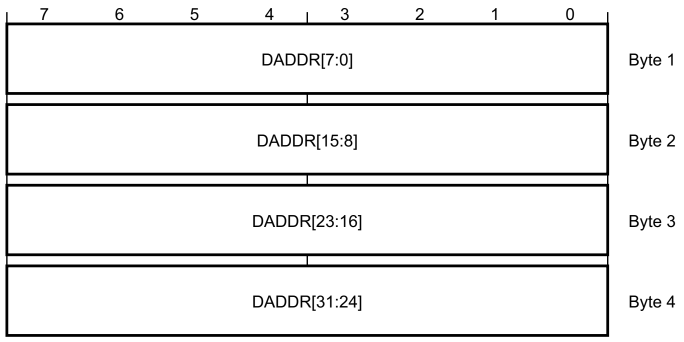
Size. The defined values of this field are:

`0b01`	Short Data Address packet.
`0b10`	Medium Data Address packet.
`0b11`	Long Data Address packet.

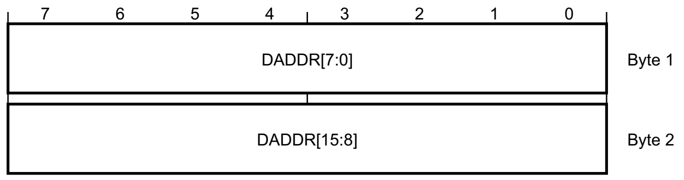
The value `0b00` encodes a Protocol packet.

Data Trace Data Address packet payload

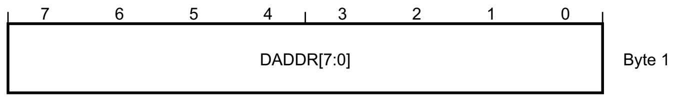
When Long Data Address packet, SS == `0b11`, the Data Trace Data Address packet payload bit assignments are:



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:

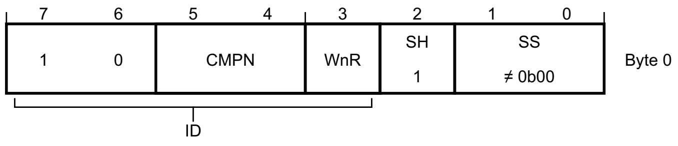
Purpose Indicates a DWT comparator generated a match, and the value that matched.

Attributes Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

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]

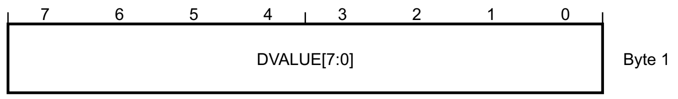
Size. The defined values of this field are:

`0b01`	Byte Data Value packet.
`0b10`	Halfword Data Value packet.
`0b11`	Word Data Value packet.

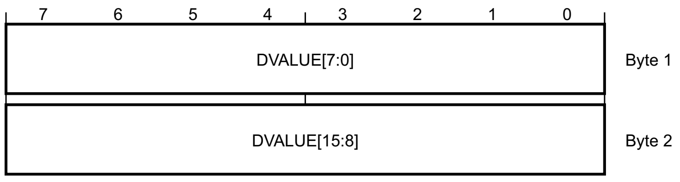
The value `0b00` encodes a Protocol packet.

Data Trace Data Value packet payload

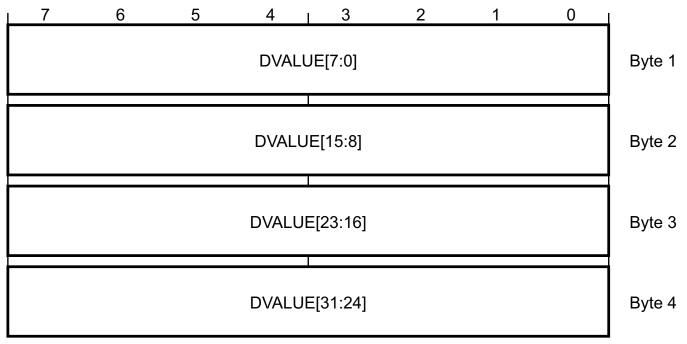
When Byte Data Value packet, SS == `0b01`, the Data Trace Data Value packet payload bit assignments are:



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.

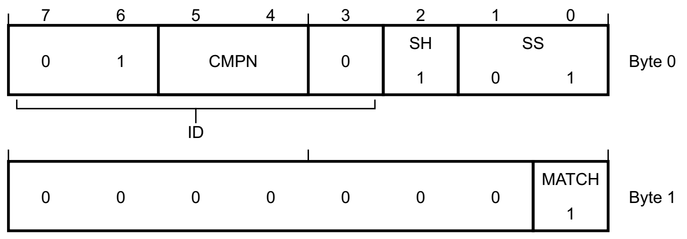
F1.2.3 Data Trace Match packet

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:

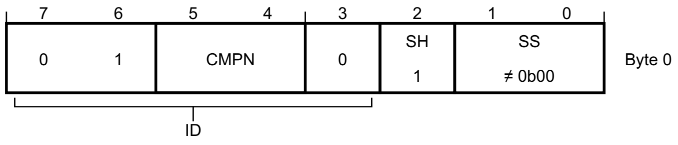
Purpose Indicates a DWT comparator generated a match, and the address of the instruction that matched. The address might be compressed. However, it is not required that Short and Medium packets are generated when the address bits match.

Attributes Multi-part Hardware source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

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]

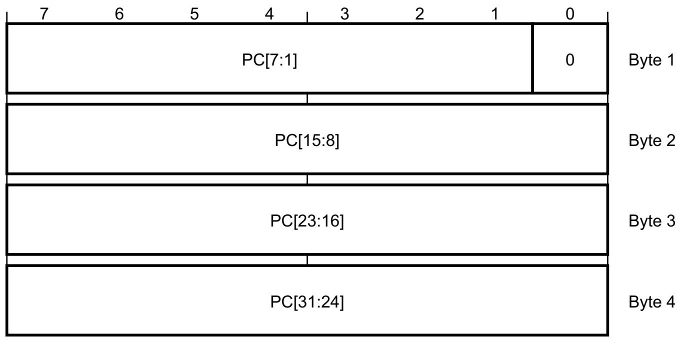
Size. The defined values of this field are:

`0b01`	Short PC Value packet.
`0b10`	Medium PC Value packet.
`0b11`	Long PC Value packet.

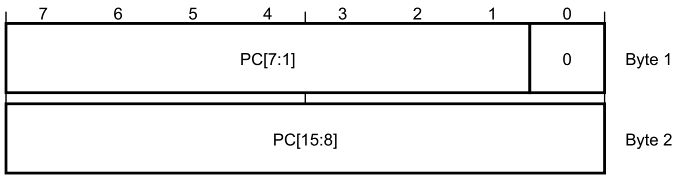
The value `0b00` encodes a Protocol packet.

Data Trace PC Value packet payload

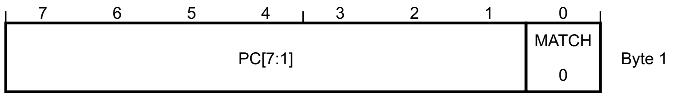
When Long PC Value packet, SS == `0b11`, the Data Trace PC Value packet payload bit assignments are:



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.

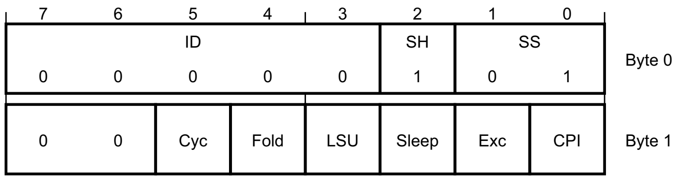
F1.2.5 Event Counter packet

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_LSUCNT 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`.

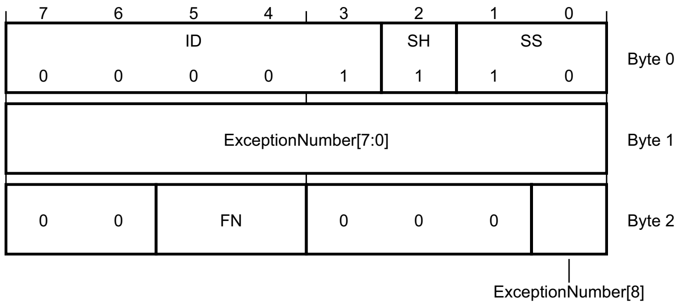
F1.2.6 Exception Trace packet

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

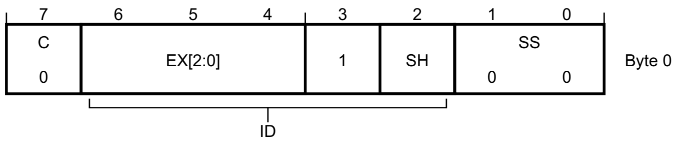
The Extension packet characteristics are:

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.

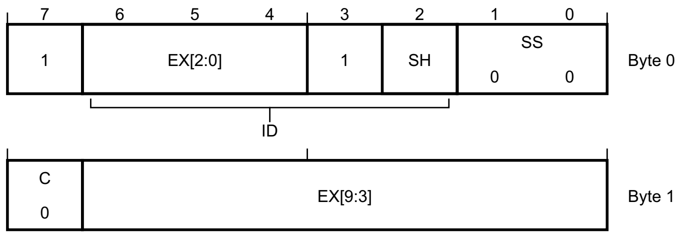
Attributes 8, 16, 24, 32, or 40-bit Protocol packet.

Field descriptions

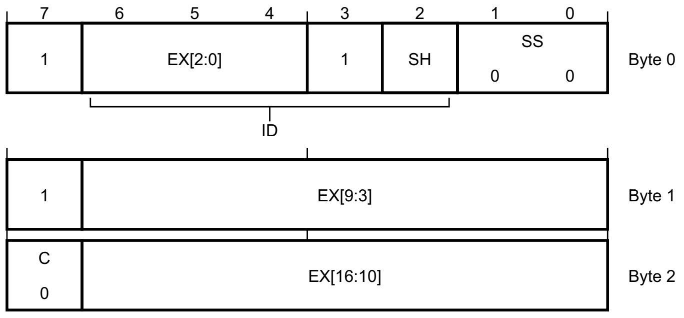
When 1-byte packet, the Extension packet bit assignments are:



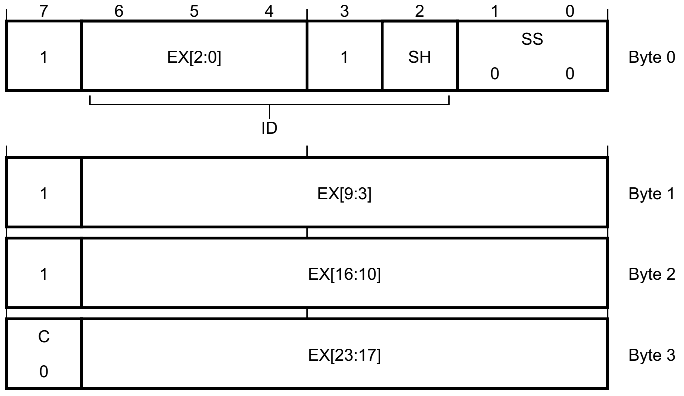
When 2-byte packet, the Extension packet bit assignments are:



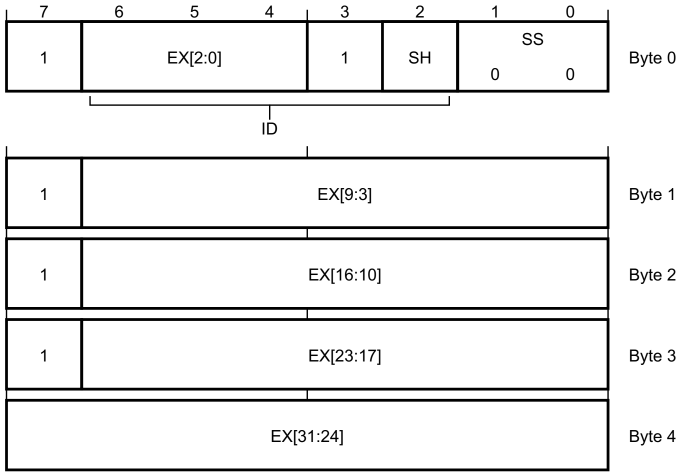
When 3-byte packet, the Extension packet bit assignments are:



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:

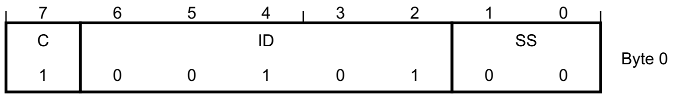
Purpose Contains the least significant bits of the global timestamp value. The ITM might compress this value if it is not generating a full timestamp by omitting significant bits if they are unchanged from the previous timestamp value.

Attributes Multi-part Protocol packet comprising:

- 8-bit header.
- 8, 16, 24, or 32-bit payload.

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]

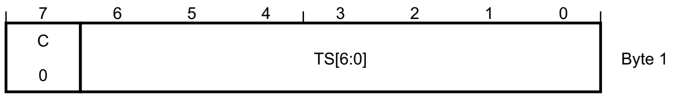
Packet type. The defined values of this field are:

`0b00` Protocol packet.

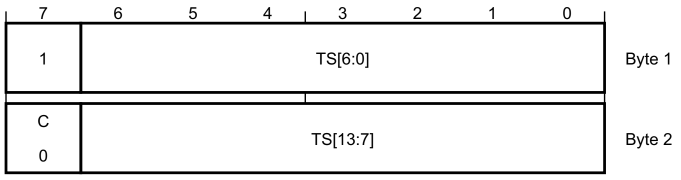
Other values encode different sizes of Hardware and Software source packets. This field reads as `0b00`.

Global Timestamp 1 packet payload

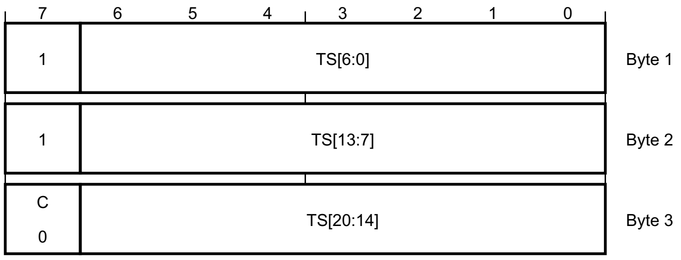
When 7-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:



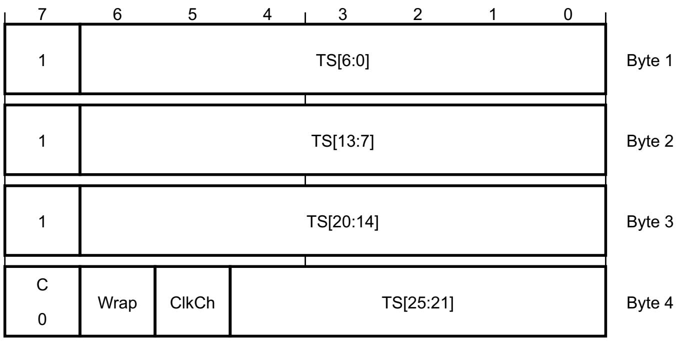
When 14-bit timestamp, the Global Timestamp 1 packet payload bit assignments are:



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.

Global Timestamp 2 packet header

The Global Timestamp 2 packet header bit assignments are:

7	6	5	4	3	2	1	0	
C	ID					SS		Byte 0
1	0	1	1	0	1	0	0	

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]

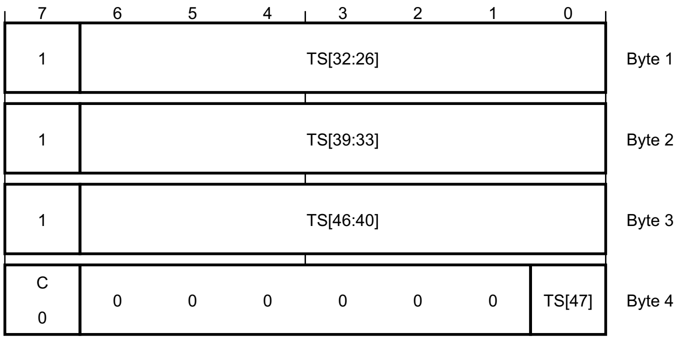
Packet type. The defined values of this field are:

`0b00` Protocol packet.

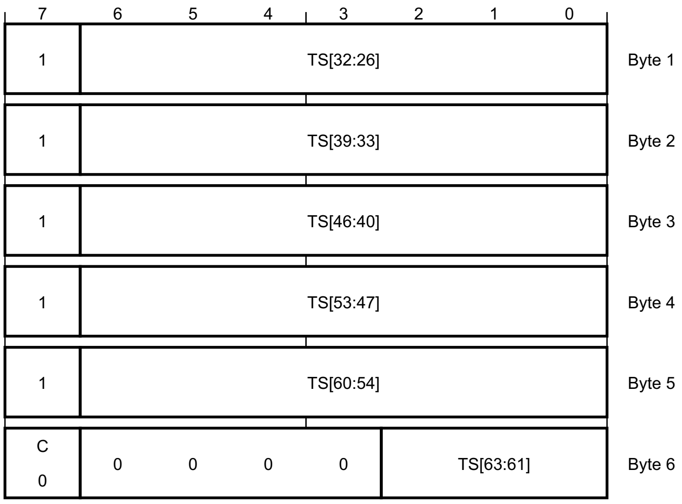
Other values encode different sizes of Hardware and Software source packets. This field reads as `0b00`.

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:

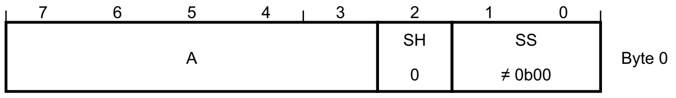
Purpose A software write to an ITM stimulus port generates an Instrumentation packet.

Attributes Multi-part Software source packet comprising:

- 8-bit header.
- 8, 16, or 32-bit payload.

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]

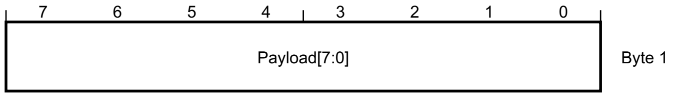
Size. The defined values of this field are:

`0b01`	Byte Instrumentation packet.
`0b10`	Halfword Instrumentation packet.
`0b11`	Word Instrumentation packet.

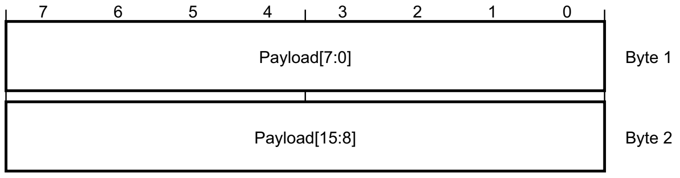
The value `0b00` encodes a Protocol packet.

Instrumentation packet payload

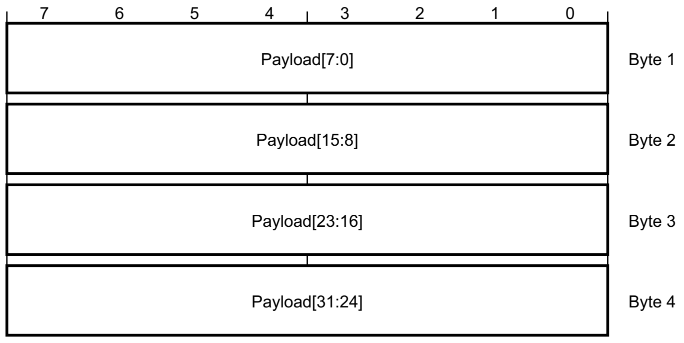
When Byte Instrumentation packet, SS == `0b01`, the Instrumentation packet payload bit assignments are:



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:

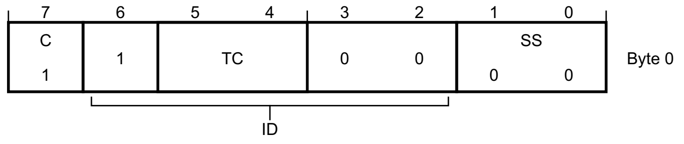
- The local timestamping scheme uses delta timestamps. Whenever the ITM outputs a Local timestamp packet, it clears its timestamp counter to zero, meaning each local timestamp value gives the interval since the generation of the previous Local timestamp packet.
- The Local Timestamp 1 packet length, 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.

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]

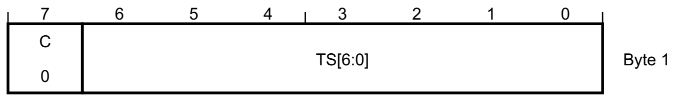
Packet type. The defined values of this field are:

`0b00` Protocol packet.

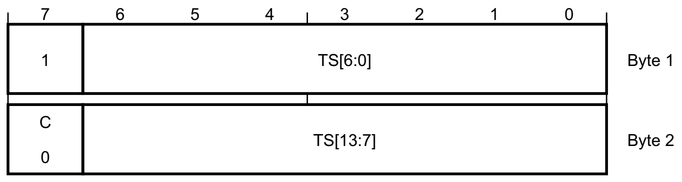
Other values encode different sizes of Hardware and Software source packets. This field reads as `0b00`.

Local Timestamp 1 packet payload

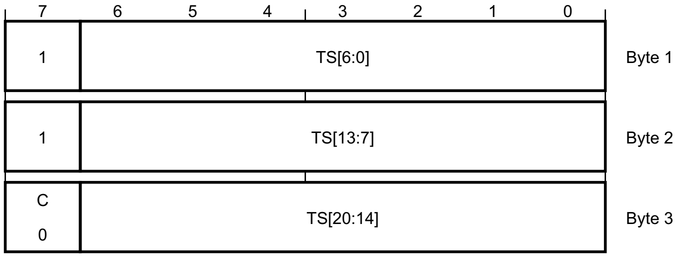
When 7-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



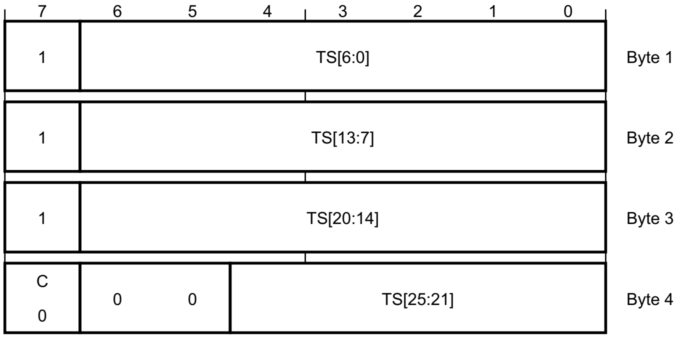
When 14-bit timestamp, the Local Timestamp 1 packet payload bit assignments are:



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.

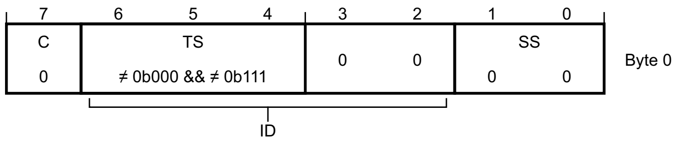
F1.2.12 Local Timestamp 2 packet

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:

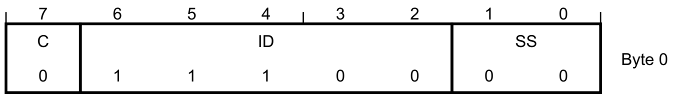
- Software writes to a Stimulus Port register when the stimulus port output buffer is full.
- The DWT attempts to generate a Hardware source packet when the DWT output buffer is full.
- The Local timestamp counter overflows.

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:

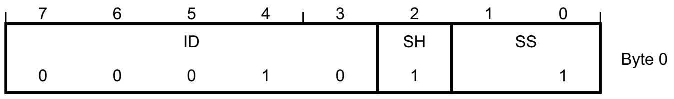
Purpose The DWT unit generates PC samples at fixed time intervals, with an accuracy of one clock cycle. The POSTCNT counter period determines the PC sampling interval. Software configures the DWT_CTRL.CYCTAP and DWT_CTRL.POSTINIT fields to determine how POSTCNT relates to DWT_CYCCNT. The DWT_CTRL.PCSAMPLENA bit enables PC sampling.

Attributes Multi-part Hardware source packet comprising:

- 8-bit header.
- 8 or 32-bit payload.

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.

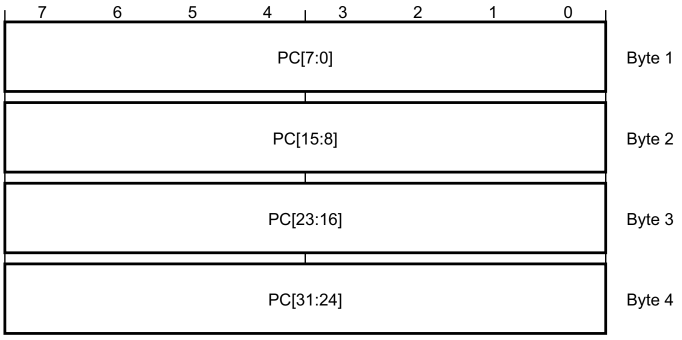
SS == `0b10` is invalid for a Periodic PC Sample packet.

The value `0b00` encodes a Protocol packet.

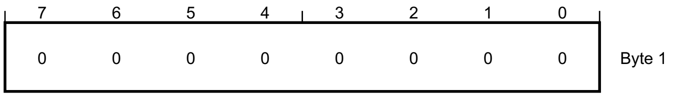
This field reads as `0bx1`.

Periodic PC Sample packet payload

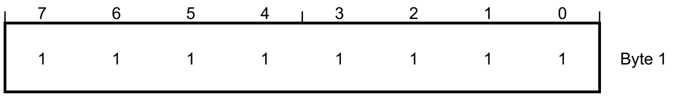
When Allowed and not sleeping, SS == `0b11`, the Periodic PC Sample packet payload bit assignments are:



When Allowed and sleeping, SS == `0b01`, the Periodic PC Sample packet payload bit assignments are:



When Prohibited, SS == `0b01`, the Periodic PC Sample packet payload bit assignments are:



PC, bytes <4:1>, when Allowed and not sleeping, SS == `0b11`

Periodic PC sample value.

Byte <1>, when Allowed and sleeping, SS == `0b01`

This field reads as `0b00000000`.

Byte <1>, when Prohibited, SS == `0b01`

This field reads as `0b11111111`.

F1.2.15 Synchronization packet

The Synchronization packet characteristics are:

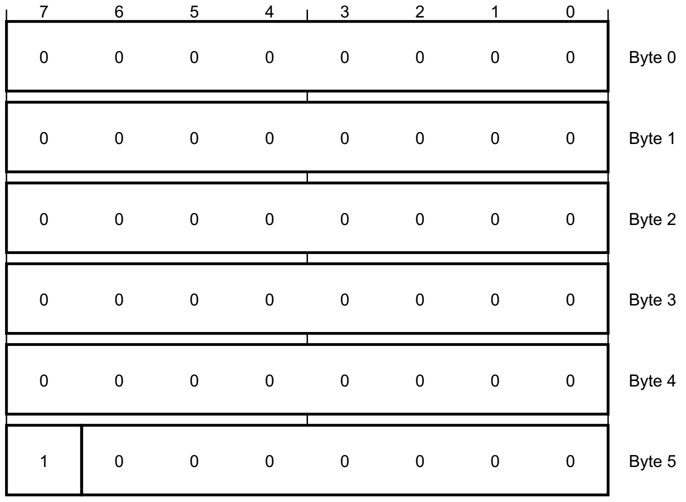
Purpose A Synchronization packet provides a unique pattern in the bit stream. Trace capture hardware can identify this pattern and use it to identify the alignment of packet bytes in the bitstream.

Attributes 48-bit Protocol packet.

A Synchronization packet is at least forty-seven 0 bits followed by single 1 bit. This section describes the smallest possible Synchronization packet.

Field descriptions

The Synchronization packet bit assignments are:



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.

APSR *See* Application Program Status Register.

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 *XPSR, APSR, IPSR, and EPSR* on page B3-45.

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.

Associativity *See* Cache associativity.

Atomicity Describes either single-copy atomicity or multi-copy atomicity. *Atomicity* on page B5-138 defines these forms of atomicity for the Arm architecture.

See also Multi-copy atomicity, Single-copy atomicity.

Attribution Unit (AU)

The combination of the Secure Attribution Unit (SAU) and the Implementation Defined Attribution Unit (IDAU).

See also Chapter B8 *The Armv8-M Protected Memory System Architecture*.

AU *See* Attribution Unit (AU).

Background state

The state of the PE before the last (previous) preemption occurred.

Banked register A register that has multiple instances, with the instance that is in use depending on the PE mode, Security state, or other PE state.

Base register A register that is specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the address that is sent to memory.

Base register Write-Back

Describes writing back a modified value to the base register used in an address calculation.

Behaves as if Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Big-endian memory

Means that, for example:

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

See also Endianness, 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 write-back granule

The maximum size of the memory that can be overwritten. In some implementations, the CTR identifies the Cache Write-Back Granule.

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 *Caches* on page B5-172.

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.

Callee-saved registers

Are registers that a called procedure must preserve. To preserve a callee-saved register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and reload it from the stack during procedure exit.

Caller-saved registers

Are registers that a called procedure is not required to preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

Coherence order

See Coherent.

Coherent Data accesses from a set of observers to a byte in memory are coherent if accesses to that byte in memory by the members of that set of observers are consistent with there being a single total order of all writes to that byte in memory by all members of the set of observers. This single total order of all to writes to that memory location is the *coherence order* for that byte in memory.

Condition code check

The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of EPSR.IT determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field

A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags The N, Z, C, and V bits of APSR, or XPSR. See *XPSR, APSR, IPSR, and EPSR* on page B3-45 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 *Conditional execution* on page C1-311.

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.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALL CAPITALS.

See also UNPREDICTABLE.

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 *Context Synchronization Event* on page B3-110.

Note

Security state transitions are not Context synchronization events.

Control dependency

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

Cross Trigger Interface

A debug component that is not part of the Armv8-M architecture.

CTI *See Cross Trigger Interface*.

DAP Debug Access Port.

Data Watchpoint and Trace (DWT)

The Data Watchpoint and Trace unit is a component of Armv8-M debug that optionally provides a number of trace, sampling, and profiling functions.

See also Data Watchpoint and Trace unit on page B12-270.

DCB *See* Debug Control Block (DCB).

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 (SCS).

Debugger In most of this manual, *debugger* refers to any agent that is performing debug. However, some parts of the manual require a more rigorous definition, and define debugger locally. See Chapter B11 *Debug*.

Deprecated Something that is present in the Arm architecture for backwards compatibility. Whenever possible software must avoid using deprecated features. Features that are deprecated but are not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

See also OPTIONAL.

Digital signal processing (DSP)

Algorithms for processing signals that have been sampled and converted to digital form. DSP algorithms often use saturated arithmetic.

Direct access A read or write of a register.

Domain In the Arm architecture, *domain* is used in the following contexts.

Shareability domain Defines a set of observers for which the Shareability attributes make the data or unified caches transparent for data accesses.

Power domain Defines a block of logic with a single, common, power supply.

Double-precision value

Consists of two consecutive 32-bit words that are interpreted as a basic double-precision floating-point number according to the *IEEE Standard for Floating-point Arithmetic*.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned

Means that the address is divisible by 8.

DSP *See* Digital signal processing (DSP).

DWT *See* Data Watchpoint and Trace (DWT).

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 *Endianness* on page B5-135.

See also Big-endian memory and Little-endian memory.

EPSR *See* Execution Program Status Register (EPSR).

ETM *See* Embedded Trace Macrocell (ETM).

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 *XPSR, APSR, IPSR, and EPSR* on page B3-45.

Execution stream

The stream of instructions that would have been executed by sequential execution of the program.

Explicit access A read from memory, or a write to memory, generated by a load or store instruction that is executed by the PE.

Flash Patch and Breakpoint Unit

The Flash Patch and Breakpoint unit supports setting breakpoints on instruction fetches.

See also *Flash Patch and Breakpoint unit* on page B12-295.

Flush-to-zero mode

A processing mode that optimizes the performance of some floating-point algorithms by replacing the denormalized operands and Intermediate results with zeros, without significantly affecting the accuracy of their final results.

FPB *See Flash Patch and Breakpoint Unit*.

General-purpose registers

The registers that the base instructions use for processing:

- The general-purpose registers are R0-R12. R13-R14 are the SP and LR, respectively. For more information, see *Registers* on page B3-41.

See also High registers, Low registers.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned
Means that the address is divisible by 2.

High registers
The general-purpose registers R8-R14. Most 16-bit T32 instructions cannot access the high registers.

Note

In some contexts, *high registers* refers to R8-R15, meaning R8-R14 and the PC.

See also General-purpose registers, Low registers.

ICI
See *Interrupt continuable instruction*.

If-Then block (IT block)
An IT block is a block of up to four instructions following an *If-Then* (`IT`) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some are the inverse of others.

Immediate and offset fields
Are unsigned unless otherwise stated.

Immediate value
A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many T32 instructions can be used with an immediate argument.

IMP
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 SMALL CAPITALS.

Implicit access
An access that is not explicit.

See also Explicit access.

Imprecise exception
An exception that is generated as the result of a system error. An imprecise exception is reported at the time that is asynchronous to the instruction that caused it.

Index register
A register that is specified in some load and store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some instruction forms permit the index register value to be shifted before the addition or subtraction.

Indirect access
A read or write of a register that is not a direct access.

For example, an indirect write to a register might occur as the side-effect of executing an instruction that does not perform a direct write to the register, or because of some operation that is performed by an external agent.

See also Direct access.

Inline literals
These are constant addresses and other data items that are held in the same area as the software itself. They are automatically generated by compilers, and can also appear in assembler code.

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.

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 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 XPSR, APSR, IPSR, and EPSR on page B3-45.

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 *Instruction set, interworking support* on page C1-320.

IPSR *See* Interrupt Program Status Register (IPSR).

ISR See *Interrupt Service Routine*.

ITM *See* Instrumentation Trace Macrocell (ITM).

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 (PoC).

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 (PoU).

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 (PoU).

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, Endianness.

Load/store architecture
An architecture where data-processing operations only operate on register contents, not directly on memory contents.

LoC *See* Level of Coherence (LoC).

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 *Lockup* on page B3-103.

LoUIS *See* Level of Unification, Inner Shareable (LoUIS).

LoUU *See* Level of Unification, uniprocessor (LoUU).

Low registers General-purpose registers R0-R7. Unlike the high registers, all T32 instructions can access the Low registers.

Memory barriers

The term memory barrier is the general term that is applied to an instruction, or sequence of instructions, that forces synchronization events by a PE regarding retiring Load/Store instructions. For more information see *Memory barriers* on page B5-150,

Memory coherency

The problem of ensuring that when a memory location is read, either by a data read or an instruction fetch, the value that is actually obtained is always the value that was most recently written to the location. This can be difficult when there are multiple possible physical locations, such as main memory and at least one of a write buffer and one or more levels of cache.

Memory hint A memory hint instruction provides advance information to memory systems about future memory accesses, without actually loading or storing any data to or from the register file. `PLD` and `PLI` are the only memory hint instructions that are defined in Armv8-M.

Memory Protection Unit (MPU)

A hardware unit whose registers provide simple control of a limited number of protection regions in memory, for more information, see Chapter B8 *The Armv8-M Protected Memory System Architecture*.

MPU *See* Memory Protection Unit (MPU).

Multi-copy atomicity

The form of atomicity that is described in *Multi-copy atomicity* on page B5-138.

See also Atomicity, Single-copy atomicity.

NaN Not a Number. A floating-point value that can be used when neither a numeric value nor an infinity is appropriate. A NaN can be a *quiet* NaN, that propagate through most floating-point operations, or a *signaling* NaN, that causes an Invalid Operation floating-point exception when used. For more information, see the *IEEE Standard for Floating-point Arithmetic*.

Non-Return-to-Zero (NRZ)

A physical layer signaling scheme that is used on asynchronous communication ports

NRZ See Non-Return-to-Zero (NRZ).

Observer A master in the system that is capable of observing memory accesses. For more information, see *Observability of memory accesses* on page B5-144.

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 SMALL CAPITALS.

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 (PE).

Physical address (PA)
An address that identifies a location in the physical memory map.

PoC *See* Point of coherency (PoC).

PoU *See* Point of unification (PoU).

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.

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.

PPB Private Peripheral Bus

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.

Pre-indexed addressing
Means that the memory address is formed in the same way as for offset addressing, but the memory address is also written back to the base register.

Privileged access
Memory systems typically 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 *XPSR, APSR, IPSR, and EPSR* on page B3-45.

Protection region
A memory region whose position, size, and other properties are defined by Memory Protection Unit registers.

Protection Unit *See* Memory Protection Unit (MPU).

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*.

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).

RAZ *See* Read-As-Zero (RAZ).

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 (RAO), 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 (RAO).

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 (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

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 (RAZ).

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 SBZP 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.

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.

Register data dependency

A register data dependency exists between a first data value and a second data value when either:

- The register that holds the first data value is used in the calculation of the second data value, and the calculation between the first data value and the second data value does not consist of either:
  - — A conditional branch whose condition is determined by the first data value.
  - — A conditional selection, move, or computation whose condition is determined by the first data value, where the input data values for the selection, move, or computation do not have a data dependency on the first data value.
- There is a register data dependency between the first data value and a third data value, and between the third data value and the second data value.

RES0

A reserved bit or field with *Should-Be-Zero-or-Preserved (SBZP)* 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 SBZP 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 SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES1

A reserved bit or field with *Should-Be-One-or-Preserved (SBOP)* 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 SMALL CAPITALS.

See also Read-As-One (RAO), RES0, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

Reserved

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.
- Bit positions that are described as reserved are:
  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES1, UNDEFINED, UNK, UNPREDICTABLE.

Return Link A value relating to the return address.

RISC Reduced Instruction Set Computer.

Rounding error The value of the rounded result of an arithmetic operation minus the exact result of the operation.

Rounding mode Specifies how the exact result of a floating-point operation is rounded to a value that is representable in the destination format. The rounding modes are defined by the *IEEE Standard for Floating-point Arithmetic.*

Saturated arithmetic

Integer arithmetic in which a result that would be greater than the largest representable number is set to the largest representable number, and a result that would be less than the smallest representable number is set to the smallest representable number. Signed saturated arithmetic is often used in DSP algorithms. It contrasts with the normal signed integer arithmetic used in Arm processors, in which overflowing results wrap around from $+2^{31}-1$ to -2^{31} or the opposite way.

SBO *See* Should-Be-One (SBO).

SBOP *See* Should-Be-One-or-Preserved (SBOP).

SBZ *See* Should-Be-Zero (SBZ).

SBZP *See* Should-Be-Zero-or-Preserved (SBZP).

Security hole A mechanism by which execution at the current level of privilege can achieve an outcome that cannot be achieved at the current or a lower level of privilege using instructions that are not UNPREDICTABLE and are not CONSTRAINED UNPREDICTABLE. The Arm architecture forbids security holes.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

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.

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.

Set *See* Cache sets.

Should-Be-One (SBO)
Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Should-Be-One-or-Preserved (SBOP)
From the introduction of the Armv8 architecture, the description *Should-Be-One-or-Preserved (SBOP)* 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 (SBZP)* is superseded by *RES0*.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 0s, it must expect an UNPREDICTABLE or CONSTRAINED UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE.

Signaling NaNs Cause an Invalid Operation exception whenever any floating-point operation receives a signaling NaN as an operand. Signaling NaNs can be used in debugging, to track down some uses of uninitialized variables.

Signed data types

Represent an integer in the range -2^{N-1} to $+2^{N-1}- 1$, using two's complement format.

Signed immediate and offset fields

Are encoded in two’s complement notation unless otherwise stated.

SIMD Single-Instruction, Multiple-Data.

Simple sequential execution

The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no Speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and Arm does not expect this model to correspond to a realistic implementation of the architecture.

Single peripheral

A single peripheral is a region of memory of an IMPLEMENTATION DEFINED size that is defined by the peripheral

Single-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*.

Single-copy atomicity

The form of atomicity that is described in *Single-copy atomicity* on page B5-138.

See also Atomicity, Multi-copy atomicity.

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 *Registers* on page B3-41.

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 (SCS).

System Control Space (SCS)

A region of the memory map that is reserved for system control and configuration registers.

See also Debug Control Block (DCB), *The System Control Space (SCS)* on page B6-192.

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*.

T32 instructions were previously called Thumb instructions.

Tail-chaining An optimization that removes unstacking and stacking operations. For more information, see *Tail-chaining* on page B3-93.

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 (TPIU).

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.

Trace Port Interface Unit (TPIU)

A component of the Arm CoreSight debug and trace solution. A TPIU provides an external interface for one or more trace sources in the processor implementation.

UAL *See* Unified Assembler Language.

Unaligned 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, UNDEFINED, UNPREDICTABLE.

UNDEFINED Indicates an instruction that generates an Undefined Instruction exception.

In body text, the term UNDEFINED is shown in SMALL CAPITALS.

See also Chapter C1 *Instruction Set Overview*.

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* 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 virtual 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 (RAO), Should-Be-One-or-Preserved (SBOP), 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 (RAZ), Should-Be-Zero-or-Preserved (SBZP), 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 UNPREDICTABLE, 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 SMALL CAPITALS.

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 SMALL CAPITALS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Unsigned data types

Represent a non-negative integer in the range 0 to $+2^N-1$, 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-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-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.

Write buffer A block of high-speed memory that optimizes stores to main memory.

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.

XPSR *See* Program Status Registers (XPSR).