Arm® Cortex®-M55 Processor Devices

Revision: r0p1

Generic User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
101273_0001_01_en

Arm® Cortex®-M55 Processor Devices

Generic User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
0001-01	31 March 2020	Non-Confidential	First early access release for r0p0

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

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

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at *http://www.arm.com/company/policies/trademarks*.

Copyright © 2020 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

www.arm.com

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Contents

Arm® Cortex®-M55 Processor Devices Generic User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

List of Figures
Arm® Cortex®-M55 Processor Devices Generic User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

List of Tables
Arm® Cortex®-M55 Processor Devices Generic User Guide

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Preface

This preface introduces the *Arm® Cortex®-M55 Processor Devices Generic User Guide*.

It contains the following:
- *About this book* on page 19.
- *Feedback* on page 22.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

About this book

This book is a generic user guide for devices that implement the Cortex®-M55 processor. Implementers of Cortex-M55 designs make several implementation choices that can affect the functionality of the device. This means that, in this book some information is described as implementation-defined, and some features are described as optional. In this book, unless the context indicates otherwise, processor refers to the Cortex-M55 processor, as supplied by Arm, and device refers to an implemented device, which is supplied by an Arm partner, that incorporates a Cortex-M55 processor. In particular, your device refers to the particular implementation of the Cortex-M55 processor that you are using. Some features of your device depend on the implementation choices made by the Arm partner that made the device.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This manual is written to help system designers, system integrators, verification engineers, and software programmers who are implementing a *System on Chip* (SoC) device based on the Cortex®-M55 processor.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction, Reference Material
This chapter provides the reference material for the introduction to a Cortex-M55 Processor User Guide.

Chapter 2 The Cortex®-M55 Processor, Reference Material
This chapter provides the reference material for the Cortex-M55 processor description in a User Guide.

Chapter 3 The Cortex®-M55 Instruction Set, Reference Material
This chapter is the reference material for the Cortex-M55 instruction set in a User Guide. It provides general information and describes a functional group of Cortex-M55 instructions. All the instructions supported by the Cortex-M55 processor are described.

Chapter 4 Cortex®-M55 Processor-level components and system registers , Reference Material
This chapter presents the reference material for the Arm Cortex-M55 processor-level components and system register descriptions in a User Guide.

Chapter 5 Reliability, Availability, and Serviceability Extension support
This chapter describes the *Reliability, Availability, and Serviceability* (RAS) features implemented in the processor.

Chapter 6 Performance Monitoring Unit Extension support
This chapter describes the *Performance Monitoring Unit* (PMU) Extension support.

Appendix A External Wakeup Interrupt Controller
This appendix describes the *External Wakeup Interrupt Controller* (EWIC) that can be optionally implemented with the processor.

Appendix B Revisions
This appendix describes the technical changes between released issues of this book.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic

Introduces special terminology, denotes cross-references, and citations.

bold

Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`

Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`

Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`

Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`

Denotes language keywords when used outside example code.

<and>

Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS

Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

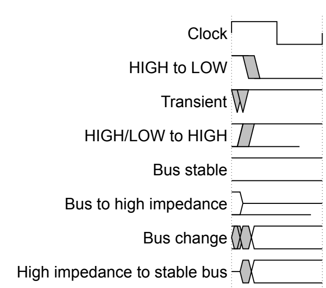


Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

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, n denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm publications

- *Arm®v8-M Architecture Reference Manual* (DDI 0553).
- *Arm® AMBA® 5 AHB Protocol Specification* (IHI 0033).
- *AMBA® APB Protocol Version 2.0 Specification* (IHI 0033).
- *AMBA® 4 ATB Protocol Specification* (IHI 0032).
- *AMBA® AXI and ACE Protocol Specification* (IHI 0022).
- *Arm® CoreSight™ System-on-Chip SoC-600 Technical Reference Manual* (100806).
- *AMBA® Low Power Interface Specification Arm® Q-Channel and P-Channel Interfaces* (IHI 0068).
- *Arm® Embedded Trace Macrocell Architecture Specification ETMv4* (ARM IHI 0064).
- *Arm® CoreSight™ Architecture Specification v3.0* (IHI 0029).
- *Arm® Debug Interface Architecture Specification, ADIv6.0* (IHI 0074).
- *Arm® Reliability, Availability, and Serviceability (RAS) Specification* (DDI 0587).
- *Arm® CoreSight™ ETM-M55 Technical Reference Manual* (101053).

The following confidential book is only available to licensees:
- *Arm® Cortex®-M55 Processor Integration and Implementation Manual* (101052).

Other publications

- *IEEE Std 1149.1-2001, Test Access Port and Boundary-Scan Architecture (JTAG)*.
- *ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Cortex-M55 Processor Devices Generic User Guide*.
- The number 101273_0001_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 1
Introduction, Reference Material

This chapter provides the reference material for the introduction to a Cortex-M55 Processor User Guide.

It contains the following sections:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.1 About this document

The material in this document is for microcontroller software and hardware engineers, including engineers with no prior experience of Arm technology. It provides the information that is required for application and system-level software development. The material does not provide information on debug components, features, or operation.

1.1.1 Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
ADD Rd, SP, #<imm>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.2 About the Cortex®-M55 processor and core peripherals

The Cortex-M55 processor is a fully synthesizeable mid-range processor that is designed for the microcontroller market. The processor offers high compute performance across both scalar and vector operations with low power consumption, fast interrupt handling, and enhanced system debug with extensive breakpoint and trace capabilities.

Other significant benefits to developers include:
- Efficient processor core, system, and memories.
- Instruction set extension for *Digital Signal Processing* (DSP) and Machine Learning applications.
- Ultra-low power consumption with integrated sleep modes.
- Platform robustness with optional integrated memory protection.
- Extended security features with optional Security Extension.
- Extended vector processing functionality with optional Armv8.1-M *M-profile Vector Extension* (MVE). Armv8.1-M MVE is also referred to as Arm Helium™ technology .

Processor implementation

The Cortex-M55 processor core has a four-stage pipeline with early completion of common arithmetic instructions and vector fetch capability on the instruction side to optimize exception entry. It also has a 32-bit instruction fetch data width and 64-bit load/store data width for efficient operation of compute workloads. The in-order processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design.

The Cortex-M55 processor provides high-end processing hardware including:

- IEEE754-compliant half-precision, single-precision, and double-precision floating-point computation.
- IEEE754-compliant Armv8.1-M MVE.
- *Single Instruction Multiple Data* (SIMD) multiplication and multiply-with-accumulate capabilities.
- Saturating arithmetic and dedicated hardware division.
- Limited dual-issue of common 16-bit instruction pairs.
- Support for exception continuable load and store multiple accesses.
- Instruction queue to decouple instruction fetching and instruction execution.
- Optimized prefetch based on fetched instruction type to minimize over-fetching and wasting dynamic power.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

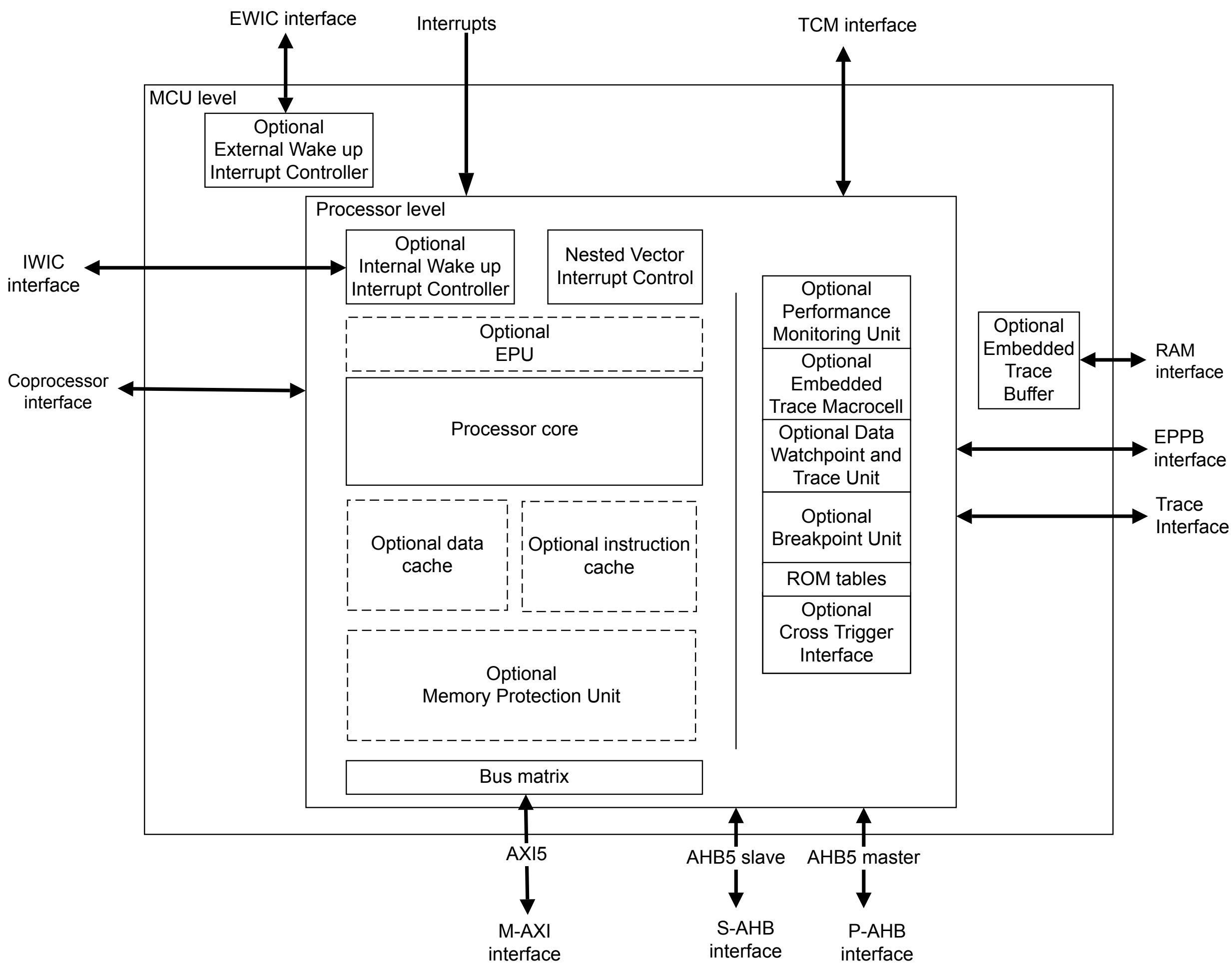


Figure 1-1 Cortex-M55 processor implementation without the Security Extension

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

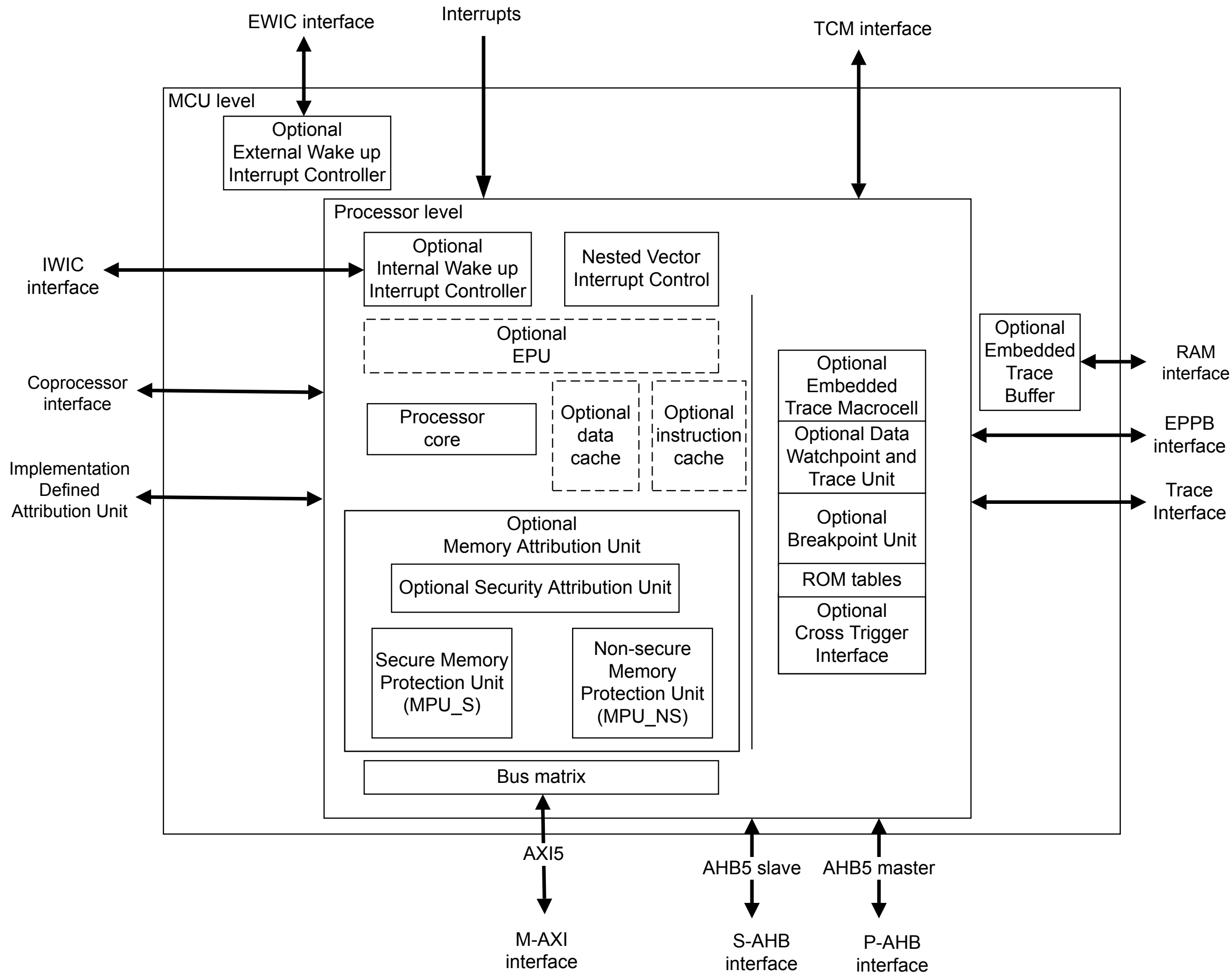


Figure 1-2 Cortex-M55 processor implementation with the Security Extension

To facilitate the design of cost-sensitive devices, the Cortex-M55 processor implements tightly-coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex-M55 processor implements the T32 instruction set based on Thumb®-2 technology, ensuring high code density and reduced program memory requirements. The Cortex-M55 processor instruction set provides the exceptional performance that is expected of a modern 32-bit architecture, with better code density than most other architectures.

The Cortex-M55 processor closely integrates a configurable *Nested Vectored Interrupt Controller* (NVIC) to deliver industry-leading interrupt performance. The NVIC includes a *non-maskable interrupt*, and provides up to 256 interrupt priority levels for other interrupts. The tight integration of the processor core and NVIC provides fast execution of *Interrupt Service Routines* (ISRs), which dramatically reduces interrupt latency. This reduced latency is achieved through:

- The hardware stacking of registers.
- The ability to suspend load multiple and store multiple operations.
- Parallel instruction-side and data-side paths.
- Tail-chaining.
- Late-arriving interrupts.

Interrupt handlers do not require wrapping in assembler code, removing any code overhead from the ISRs. The tail-chain optimization also significantly reduces the overhead when switching from one ISR to another.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To optimize low-power designs, the NVIC supports different sleep modes, including a deep sleep function that enables the entire device to be rapidly powered down while still retaining program state.

The MCU vendor determines the reliability features configuration, therefore reliability features can differ across different devices and families.

To increase instruction throughput, the Cortex-M55 processor can execute certain pairs of 16-bit instructions simultaneously. This is called dual issue.

1.2.1 System-level interface

The Cortex-M55 processor provides multiple interfaces using the following Arm protocols to provide high speed, low latency memory accesses.

- AMBA 5 AXI.
- AMBA 5 AHB.
- AMBA 4 ATB.
- AMBA 4 APB.

1.2.2 Security Extension

The Security Extension adds security through code and data protection features.

A processor with the Security Extension supports both Non-secure and Secure states, which are orthogonal to the traditional thread and handler modes. The four modes of operation are:

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

When the Security Extension is implemented, the following happens:
- Regions of memory can be configured to be only accessible by Secure code.
- Some system and debug features can only be accessed via Secure code
- The processor resets into Secure state.
- Some registers are banked between Security states. There are two separate instances of the same register, one in Secure state and one in Non-secure state.
- The architecture allows the Secure state to access the Non-secure versions of banked registers.
- Interrupts can be configured to target one of the two Security states.
- Some faults are banked between Security states or are configurable.

The processor also supports security gating on TCM interfaces.

1.2.3 Integrated configurable debug

The Cortex-M55 processor implements a complete hardware debug solution. This provides high system visibility of the processor and memory through either a traditional JTAG port or a 2-pin *Serial Wire Debug* (SWD) port that is ideal for microcontrollers and other small package devices. The MCU vendor determines the debug feature configuration, therefore debug features can differ across different devices and families.

The processor provides instruction and data trace and profiling support. To enable simple and cost-effective profiling of the resulting system events, a *Serial Wire Viewer* (SWV) can export a stream of software-generated messages, data trace, and profiling information through a single pin.

When implemented, debuggers can use:

- The *Breakpoint Unit* (BPU), which supports four or eight hardware breakpoint comparators.
- The *Data Watchpoint and Trace* (DWT), which supports four or eight watchpoint comparators.
- The Armv8.1-M *Digital Signal Processing* (DSP) debug extension for analysis of signal processing and compute-based software.
- Monitor mode for self-hosted debug.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Debug functionality also supports the following:

- The *Digital Signal Processing* (DSP) debug extension for analysis of signal processing and compute-based software.
- Full access to the memory map and registers through a 32-bit *Debug AHB* (D-AHB) interface.
- An *Instrumentation Trace Macrocell* (ITM) for software-driven `printf` debugging which can be linked to the DWT.
- An implementation of the Armv8.1-M *Performance Monitoring Unit* (PMU).
- An *Embedded Trace Macrocell* (ETM) which supports complete instruction trace. It implements the ETMv4.5 architecture, including support for tracing the Armv8.1-M *M-profile Vector Extension* (MVE) features. Data trace is not supported.
- Access control that prevents unauthorized debug or trace of Secure state or memory, including support for the Armv8.1-M Unprivileged Debug Extension for fine-grain control of debug access to the processor.

The Cortex-M55 processor always implements the debug monitor mode functionality for self-hosted debug.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.2.4 Processor features and benefits summary

The Cortex-M55 processor benefits include tight integration of system peripherals that reduces area and development costs, T32 instruction set that combines high code density with 32-bit performance, and IEEE754-compliant half-precision, single-precision, and double-precision floating-point functionality and Armv8.1-M *M-profile Vector Extension* (MVE).

Other processor features and benefits are:
- Power control optimization of system components.
- Integrated sleep modes for low power consumption.
- Security Extension.
- Fast code execution permits slower processor clock or increases sleep mode time.
- Hardware integer division and fast multiply accumulate for digital signal processing.
- Saturating arithmetic for signal processing.
- Deterministic, high-performance interrupt handling for time-critical applications.
- *Memory Protection Unit* (MPU) and *Security Attribution Unit* (SAU) for safety-critical applications.
- Extensive debug and trace capabilities.

1.2.5 Processor core peripherals

The processor has the following core peripherals:

Nested Vectored Interrupt Controller

The *Nested Vectored Interrupt Controller* (NVIC) is an embedded interrupt controller that supports low-latency interrupt processing.

System Control Space

The *System Control Space* (SCS) is the programmer's model interface to the processor. It provides system implementation information and system control.

System timer

The system timer, SysTick, is a 24 bit count-down timer. Use this as a *Real Time Operating System* (RTOS) tick timer or as a simple counter. In an implementation with the Security Extension, there are two SysTicks, one Secure and one Non-secure.

The system timer, SysTick, is a 24 bit count-down timer. Use this as a *Real Time Operating System* (RTOS) tick timer or as a simple counter. There are two SysTicks, one Secure and one Non-secure.

Security Attribution Unit

The *Security Attribution Unit* (SAU) improves system security by defining security attributes for different regions. It provides up to eight different regions.

Memory Protection Unit

The MPU improves system reliability by defining the memory attributes for different memory regions. It provides up to 16 different regions, and an optional predefined background region. When the Security Extension is included, there can be two MPUs, one Secure and one Non-secure. Each MPU can define memory attributes independently.

Extension Processing Unit

The *Extension Processing Unit* (EPU) provides IEEE754-compliant half-precision, single-precision, and double-precision floating-point values and Armv8.1-M *M-profile Vector Extension* (MVE).

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

1.3 Arm®v8.1-M enablement content

The following list of documents, while not specific to this product, contain important information that can assist you in developing your Cortex-M55 processor. The following documents are applicable for Armv8-M and Armv8.1-M architecture.

- *Arm®v8-M Processor Debug* (100734).
- *ACLE Extensions for Arm®v8-M* (100739).
- *Fault Handling and Detection* (100691).
- *Arm® Synchronization Primitives Development Article* (ID012816).
- *Arm®v8-M Exception Handling* (100701).
- *Memory Protection Unit for Arm®v8-M based platforms* (100699).
- *Arm®v8-M Architecture Reference Manual* (DDI 0553).
- *TrustZone® technology for Arm®v8-M Architecture* (100690).

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 2
The Cortex®-M55 Processor, Reference Material

This chapter provides the reference material for the Cortex-M55 processor description in a User Guide.

It contains the following sections:

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1 Programmer's model

The programmer's model describes the modes, privilege levels, Security states, stacks and core registers available for software execution.

2.1.1 Processor modes and privilege levels for software execution

There are two operating modes and two privilege levels.

Modes

The two modes are:

Thread mode

- Intended for applications.
- Execution can be privileged or unprivileged.
- The processor enters Thread mode out of reset and returns to Thread mode on completion of an exception handler.

Handler mode

- Intended for OS kernel and associated functions, that manage system resources.
- Execution is always privileged.
- All exceptions cause entry into Handler mode.

Privilege levels

There are two privilege levels:

Unprivileged

Software has limited access to system resources.

Privileged

Software has full access to system resources, subject to security restrictions.

In Thread mode, the CONTROL.nPRIV controls whether software execution is privileged or unprivileged. In Handler mode, software execution is always privileged.

Only privileged software can write to the CONTROL register to change the privilege level for software execution in Thread mode. Unprivileged software can use the `SVC` instruction to make a *Supervisor Call* to transfer control to privileged software.

2.1.2 Security states

There are two Security states, Secure and Non-secure. When the Security Extension is implemented, there are two Security states, Secure and Non-secure.

Security states are orthogonal to mode and privilege. Therefore each Security state supports execution in both modes and both levels of privilege.

The processor always resets into Secure state. Registers in the *System Control Space* (SCS) are banked across Secure and Non-secure state, with the Non-secure register view available at an aliased address to Secure state.

Security states are orthogonal to mode and privilege. Therefore each Security state supports execution in both modes and both levels of privilege.

When the Security Extension is implemented, the processor always resets into Secure state. Registers in the *System Control Space* (SCS) are banked across Secure and Non-secure state, with the Non-secure register view available at an aliased address to Secure state.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.1.3 Core registers

The following figures and tables illustrate the core registers of the Cortex-M55 processor:

- Without the Security Extension.
- With the Security Extension.

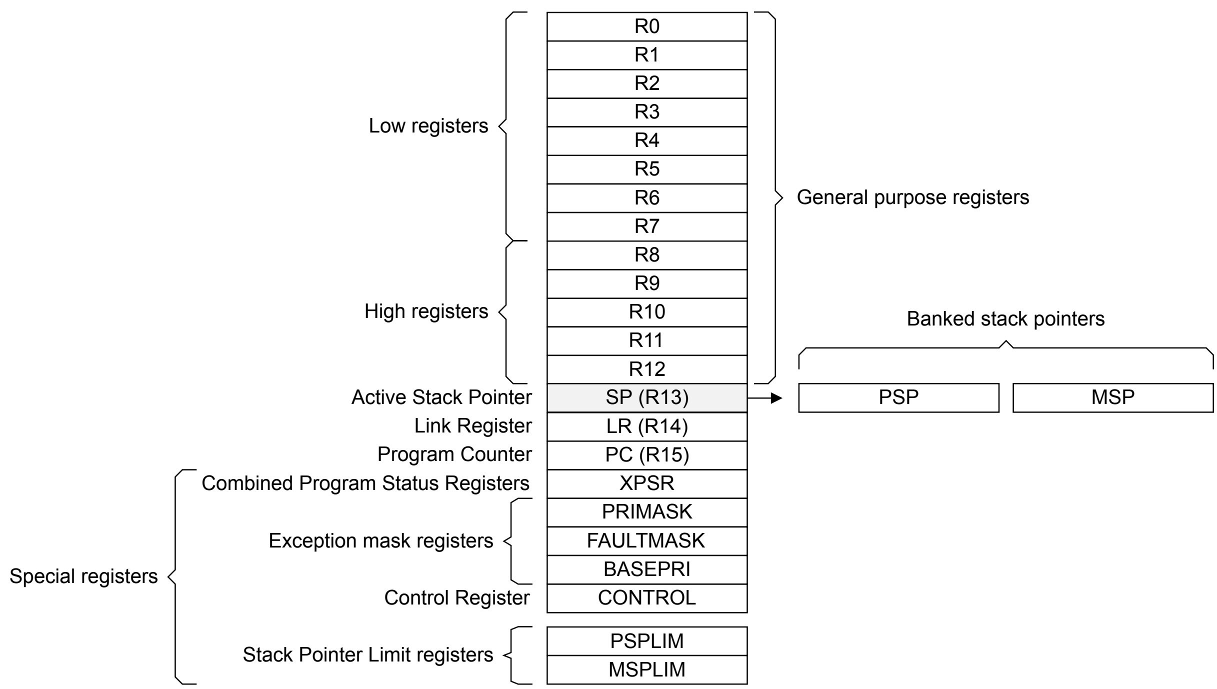


Figure 2-1 Core registers without the Security Extension

Table 2-1 Core register set summary without the Security Extension

Name	Type [a]	Required privilege [b]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	*General-purpose registers* on page 2-36
MSP	RW	Either	-[c]	*Stack Pointer* on page 2-36
PSP	RW	Either	UNKNOWN	
LR	RW	Either	`0xFFFFFFFF`	*Link Register* on page 2-39
PC	RW	Either	-[c]	*Program Counter* on page 2-39
XPSR (includes APSR, IPSR, and EPSR)	RW	Either	-[d]	*Combined Program Status Register* on page 2-39
APSR	RW	Either	UNKNOWN	*Application Program Status Register* on page 2-40
IPSR	RO	Privileged	`0x00000000`	*Interrupt Program Status Register* on page 2-40
EPSR	RO	Privileged	-[d]	*Execution Program Status Register* on page 2-42.
RETPSR	RO	Privileged	UNKNOWN	*Combined Exception Return Program Status Register, RETPSR* on page 2-42
PRIMASK	RW	Privileged	`0x00000000`	*Priority Mask Register* on page 2-44

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-1 Core register set summary without the Security Extension (continued)

Name	Type [a]	Required privilege [b]	Reset value	Description
FAULTMASK	RW	Privileged	`0x00000000`	*Fault Mask Register* on page 2-45
BASEPRI	RW	Privileged	`0x00000000`	*Base Priority Mask Register* on page 2-45
CONTROL	RW	Privileged	`0x00000000`	*CONTROL register* on page 2-46
PSPLIM	RW	Privileged	`0x00000000`	*Stack limit registers* on page 2-38
MSPLIM	RW	Privileged		
VPR	RW	Privileged	`0x00XXXXXX`	*Vector Predication Status and Control Register, VPR* on page 2-48

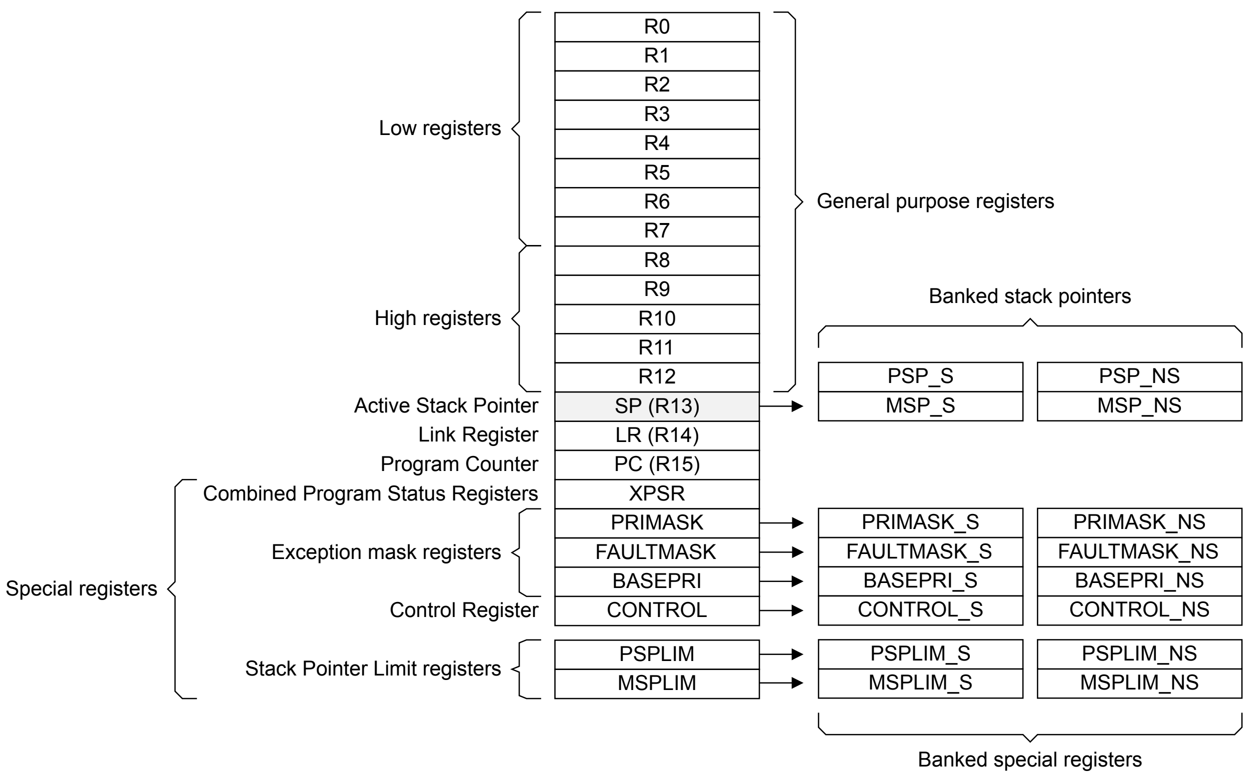


Figure 2-2 Core registers with the Security Extension

a Describes access type during program execution in Thread mode and Handler mode. Debug access can differ.
b An entry of Either means privileged and unprivileged software can access the register.
c Soft reset to the value retrieved by the reset handler
d Bit[24] is the T-bit and is loaded from bit[0] of the reset vector. All other bits are reset to 0.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-2 Core register set summary with the Security Extension

Name	Type [a]	Required privilege[b]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	*General-purpose registers* on page 2-36.
MSP_S	RW	Either	-[c]	*Stack Pointer* on page 2-36
MSP_NS		Either		
PSP_S	RW	Either	UNKNOWN	
PSP_NS		Either		
LR	RW	Either	UNKNOWN	*Link Register* on page 2-39
PC	RW	Either	-[c]	*Program Counter* on page 2-39
XPSR (includes APSR, IPSR, and EPSR)	RW	Either	-[d]	*Combined Program Status Register* on page 2-39
APSR	RW	Either	UNKNOWN	*Application Program Status Register* on page 2-40.
IPSR	RO	Privileged	`0x00000000`	*Interrupt Program Status Register* on page 2-40
EPSR	RO	Privileged	-[d]	*Execution Program Status Register* on page 2-42
RETPSR	RW	Privileged	UNKNOWN	*Combined Exception Return Program Status Register, RETPSR* on page 2-42
PRIMASK_S	RW	Privileged	`0x00000000`	*Priority Mask Register* on page 2-44
PRIMASK_NS		Privileged	`0x00000000`	
FAULTMASK_S	RW	Privileged	`0x00000000`	*Fault Mask Register* on page 2-45
FAULTMASK_NS		Privileged	`0x00000000`	
BASEPRI_S	RW	Privileged	`0x00000000`	*Base Priority Mask Register* on page 2-45
BASEPRI_NS		Privileged	`0x00000000`	
CONTROL_S	RW	Privileged	`0x00000000`	*CONTROL register* on page 2-46
CONTROL_NS		Privileged	`0x00000000`	
MSPLIM_S	RW	Privileged	`0x00000000`	*Stack limit registers* on page 2-38
MSPLIM_NS		Privileged	`0x00000000`	
PSPLIM_S	RW	Privileged	`0x00000000`	
PSPLIM_NS		Privileged	`0x00000000`	
VPR	RW	Privileged	`0x00XXXXXX`	*Vector Predication Status and Control Register, VPR* on page 2-48

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack Pointer

The *stack pointer* (SP) is register R13.

The processor uses a full descending stack, meaning the Stack Pointer holds the address of the last stacked item in memory. When the processor pushes a new item onto the stack, it decrements the Stack Pointer and then writes the item to the new memory location.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When Security Extension is implemented, software must initialize MSP_NS.

Table 2-3 Stack pointer register without the Security Extension

Stack	Stack pointer register
Main	MSP
Process	PSP

In Thread mode, the CONTROL.SPSEL bit indicates the stack pointer to use.

0 *Main stack pointer* (MSP). This is the reset value.

1 *Process stack pointer* (PSP)

Table 2-4 Stack pointer register with the Security Extension

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

In Non-secure Thread mode, the CONTROL_NS.SPSEL bit indicates the stack pointer to use:

0 *Main stack pointer* (MSP_NS). This is the reset value.

1 *Process stack pointer* (PSP_NS).

In Non-secure Handler mode, the MSP_NS is always used.

In Secure Thread mode, the CONTROL_S.SPSEL bit indicates the stack pointer to use:

0 *Main stack pointer* (MSP_S). This is the reset value.

1 *Process stack pointer* (PSP_S).

In Secure Handler mode, the MSP_S is always used.

The current Security state of the processor determines whether the Secure or Non-secure stacks are used.

To ensure that stacks do not overrun, the processor has stack limit check registers that can be programmed to define the bounds for each of the implemented stacks.

The bit assignments for the SP register is as follows:

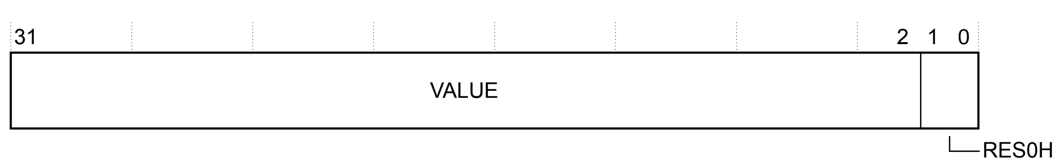


Figure 2-3 SP register bit assignments

The following table describes the SP register bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-5 SP register bit assignments

Bits	Name	Function
[31:2]	VALUE	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.
[1:0]	-	Reserved, RES0H. A reserved bit or field with *Should-Be-Zero-or-Preserved* (SBZP). This behavior uses the Hardwired to 0 subset of the RES0 definition.

Stack limit registers

The stack limit registers define the lower limit for the corresponding stack. The processor raises an exception on most instructions that attempt to update the stack pointer below its defined limit.

If the Security Extension is not implemented, the Cortex-M55 processor has two stack limit registers, as the following table shows.

Table 2-6 Stack limit registers without the Security Extension

Stack	Stack limit register
Main	MSPLIM
Process	PSPLIM

If the Security Extension is implemented, the Cortex-M55 processor has four stack limit registers, as the following table shows.

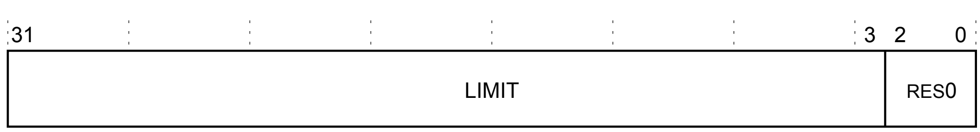
Table 2-7 Stack limit registers with the Security Extension

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

Note

The four stack limit registers are banked between Security states.

The bit assignments for the MSPLIM and PSPLIM registers are as follows:



The following table describes the MSPLIM and PSPLIM register bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-8 MSPLIM and PSPLIM register bit assignments

Bits	Name	Function
[31:3]	LIMIT	Main stack limit or process stack limit address for the selected Security state. Limit address for the selected stack pointer.
[2:0]	-	Reserved, RES0.

Link Register

The *Link Register* (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the processor sets the LR value to `0xFFFFFFFF`.

Program Counter

The *Program Counter* (PC) is register R15. It contains the current program address.

On reset, the processor loads the PC with the value of the reset vector defined in the vector table.

On reset, the processor loads the PC with the value of the reset vector defined in the Secure vector table.

If the Security Extension is implemented, the processor loads the PC with the value of the reset vector defined in the Secure vector table on reset. If the Security Extension is not implemented, the processor still loads the PC with the value of the reset vector in the vector table on reset.

Combined Program Status Register

The Combined Program Status Register (XPSR) consists of the *Application Program Status Register* (APSR), *Interrupt Program Status Register* (IPSR), and *Execution Program Status Register* (EPSR).

These registers are mutually exclusive bit fields in the 32-bit PSR. The bit assignments are as follows:

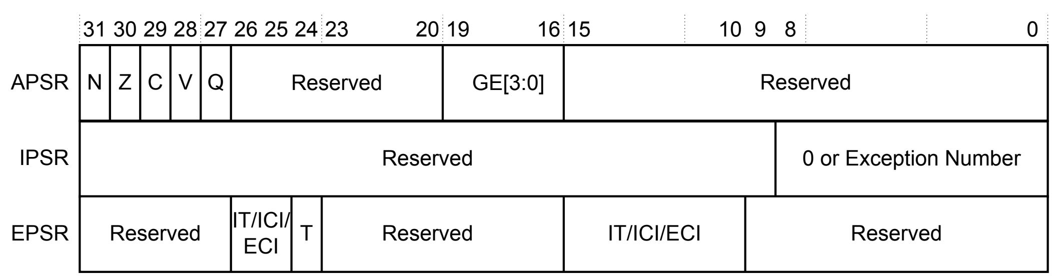


Figure 2-4 XPSR bit assignments

Access these registers individually or as a combination of any two or all three registers, using the register name as an argument to the `MSR` or `MRS` instructions. For example:

- Read all the registers using `PSR` with the `MRS` instruction.
- Write to the APSR N, Z, C, V, and Q bits using `APSR_nzcvq` with the `MSR` instruction.

The PSR combinations and attributes are:

Table 2-9 XPSR register combinations

Register	Type	Combination
XPSR	RW[e, f]	APSR, EPSR, and IPSR
IEPSR	RO[f]	EPSR and IPSR

e The processor ignores writes to the IPSR bits.
f Reads of the EPSR bits return zero, and the processor ignores writes to these bits.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-9 XPSR register combinations (continued)

Register	Type	Combination
IAPSR	RW[e]	APSR and IPSR
EAPSR	RW[f]	APSR and EPSR

See the `MRS` and `MSR` instruction descriptions for more information about how to access the Program Status Registers.

Application Program Status Register

The APSR contains the current state of the condition flags from previous instruction executions.

The APSR bit assignments are as follows:

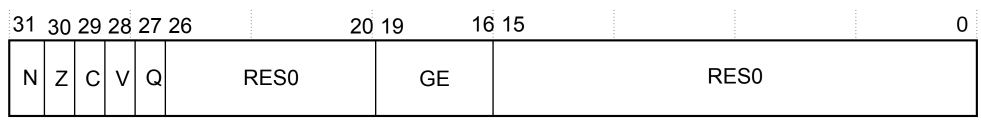


Figure 2-5 APSR bit assignments

The following table describes the APSR register bit assignments.

Table 2-10 APSR bit assignments

Bits	Name	Function
[31]	N	Negative flag.
[30]	Z	Zero flag.
[29]	C	Carry or borrow flag.
[28]	V	Overflow flag.
[27]	Q	DSP overflow and saturation flag.
[26:20]	-	Reserved.
[19:16]	GE[3:0]	Greater than or Equal flags. See *3.13.12 SEL* on page 3-144 for more information.
[15:0]	-	Reserved.

Interrupt Program Status Register

The IPSR contains the exception number of the current ISR.

The following figure shows the IPSR bit assignments if 0-479 interrupts are implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

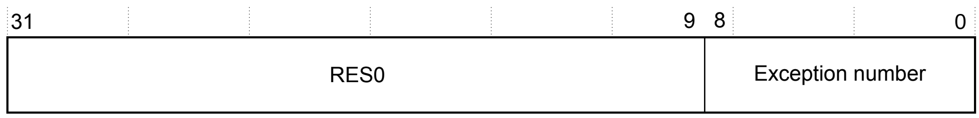


Figure 2-6 IPSR bit assignments if 0-479 interrupts are implemented

The following table shows the IPSR bit assignments.

Table 2-11 IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved.
[8:0]	Exception number	This is the number of the current exception: 0 = Thread mode. 1 = Reset. 2 = NMI. 3 = HardFault. 4 = MemManage. 5 = BusFault. 6 = UsageFault 7 = SecureFault 8-10 = Reserved. 7-10 = Reserved. 11 = SVCall. 12 = DebugMonitor. 13 = Reserved. 14 = PendSV. 15 =SysTick 16 = IRQ0. . . . 495 = IRQ479.

The active bits in the Exception number field depend on the number of interrupts implemented.

0-47 interrupts = [5:0].

48-111 interrupts = [6:0].

112-239 interrupts = [7:0].

240-479 interrupts = [8:0].

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Execution Program Status Register

The EPSR contains the Thumb state bit and the execution state bits for the *If-Then* (IT) instruction, *Interruptible-Continuable Instruction* (ICI) field for an interrupted load multiple or store multiple instruction, and Exception continuation flags for beat-wise vector instructions (ECI) field for beats of the in-flight instructions have completed.

The following table shows the EPSR bit assignments.

Table 2-12 EPSR bit assignments

Bits	Name	Function
[31:27]	-	Reserved, RES0
[26:25], [15:10]	ICI	Interruptible-continuable instruction bits, see *Interruptible-continuable instructions* on page 2-43
[26:25], [15:10]	IT	Indicates the execution state bits of the `IT` instruction, see *3.19.5 IT* on page 3-215
[26:25], [11:10], [15:12]	ECI	Exception continuation flags for beat-wise vector instructions. This field encodes which beats of the in-flight instructions have completed.
[24]	T	Thumb state bit, see *Thumb® state* on page 2-44
[23:16]	-	Reserved, RES0
[9:0]	-	Reserved, RES0

Attempts to read the EPSR directly through application software using the `MSR` instruction always return zero. Attempts to write the EPSR using the `MSR` instruction in application software are ignored.

Combined Exception Return Program Status Register, RETPSR

The RETPSR contains the 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.

The following table shows the RETPSR bit assignments.

Table 2-13 RETPSR bit assignments

Bits	Name	Function
[31]	N	Negative flag. Value corresponding to APSR.N.
[30]	Z	Zero flag. Value corresponding to APSR.Z.
[29]	C	Carry flag. Value corresponding to APSR.C.
[28]	V	Overflow flag. Value corresponding to APSR.V.
[27]	Q	Saturate flag. Value corresponding to APSR.Q.
[24]	T	T32 state. Value corresponding to EPSR.T.
[23:21]	-	Reserved, RES0
[20]	SFPA	Secure Floating-point active. Value corresponding to CONTROL.SFPA.
[19:16]	GE	Greater-than or equal flag. Value corresponding to APSR.GE.
[15:10], [26:25]	IT, when {RETPSR[26:25], RETPSR[11:10]} != 0	If-then flags. Value corresponding to EPSR.IT.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-13 RETPSR bit assignments (continued)

Bits	Name	Function
[26:25], [15:10]	ICI, when {RETPSR[26:25], RETPSR[11:10]} == 0, and a multi-cycle load or store instruction was in progress when the exception was taken	Interrupt continuation flags. Value corresponding to EPSR.ICI.
[26:25], [11:10], [15:12]	ECI, when {RETPSR[26:25], RETPSR[11:10]} == 0, and beat-wise vector instructions were in progress when the exception was taken	Exception continuation flags for beat-wise vector instructions. Value corresponding to EPSR.ECI.
[9]	SPREALIGN	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.
[8:0]	Exception	Exception numbers. Value corresponding to IPSR.Exception.

Interruptible-continuable instructions

When an interrupt occurs during the execution of an `LDM`, `STM`, `PUSH`, `POP`, `VLDM`, `VSTM`, `VPUSH`, or `VPOP` instruction, the processor can stop the load multiple or store multiple instruction operation temporarily, storing the next register operand in the multiple operation to be transferred into EPSR[15:12].

After servicing the interrupt, the processor resumes execution of the load or store multiple, starting at the register stored in EPSR[15:12].

When the EPSR holds ICI execution state, bits[26:25,11:10] are zero.

Note

There might be cases where the processor cannot pause and resume load or store multiple instructions in this way. When this happens, the processor restarts the instruction from the beginning on return from the interrupt. As a result, your software should never use load or store multiple instructions to memory that is not robust to repeated accesses.

If-Then block

The If-Then block contains up to four instructions following an `IT` instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others.

Note

Interruptible-continuable operation is not supported when the load multiple or store multiple instructions are located inside an If-Then block. In these cases, the processor can take an interrupt part-way through the load or store multiple instruction, restarting it from the beginning on return from the interrupt.

Exception continuation flags for beat-wise vector instructions

When EPSR[26:25] and EPSR[11:10] are 0, and beat-wise vector instructions are in progress, the EPSR.ECI field encodes which beats of the in-flight instructions have completed.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In the following enumeration, the letters correspond to the instructions at the return address and beyond, while the numbers correspond to the beats of those instructions that have been completed. For example, the sequence A0 A1 A2 B0 means the first three beats of the instruction at the return address, plus the first beat of the instruction at the return address +4 have been completed. The field ECI[7:0] is equivalent to EPSR[26:25,11:10,15:12].

`0b00000000`	No completed beats.
`0b00000001`	Completed beats: A0
`0b00000010`	Completed beats: A0 A1
`0b00000011`	Reserved.
`0b00000100`	Completed beats: A0 A1 A2
`0b00000101`	Completed beats: A0 A1 A2 B0
`0b0000011X`	Reserved.
`0b00001XXX`	Reserved.

Thumb® state

The Cortex-M55 processor only supports execution of instructions in Thumb state.

The following can modify the T bit in the EPSR:
- Instructions `BLX`, `BX`, `LDR pc, []`, and `POP{PC}`.
- Restoration from the stacked xPSR value on an exception return.
- Bit[0] of the exception vector value on an exception entry or reset.

Attempting to execute instructions when the T bit is 0 results in a fault or lockup. See 2.5.4 Lockup on page 2-80 for more information.

Exception mask registers

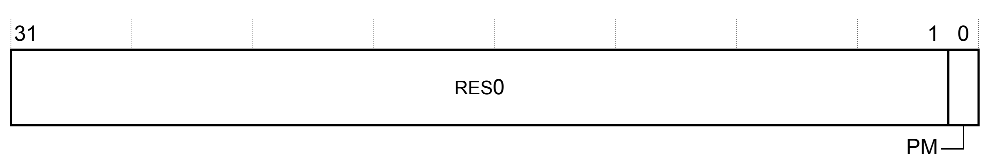
The exception mask registers disable the handling of exceptions by the processor. For example, you might want to disable exceptions when running timing critical tasks.

To access the exception mask registers use the `MSR` and `MRS` instructions, or the `CPS` instruction to change the value of PRIMASK.PM or FAULTMASK.FM.

Priority Mask Register

The PRIMASK register is intended to disable interrupts by preventing activation of all exceptions with configurable priority in the current Security state.

The bit assignments for the PRIMASK register are as follows:



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-14 PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, RES0.
[0]	PM	If the Security Extension is implemented, then setting the Secure PRIMASK to 1 raises the execution priority to 0. If the Security Extension is implemented and if AIRCR.PRIS is clear, setting the Non-secure PRIMASK to 1 raises the execution priority to 0. If the Security Extension is implemented and if AIRCR.PRIS is set, setting the Non-secure PRIMASK to 1 raises the execution priority to `0x80`

Fault Mask Register

The FAULTMASK register prevents activation of all exceptions with configurable priority and also some exceptions with fixed priority depending on the value of AIRCR.BFHFNMINS and AIRCR.PRIS.

The bit assignments for the FAULTMASK register are as follows:

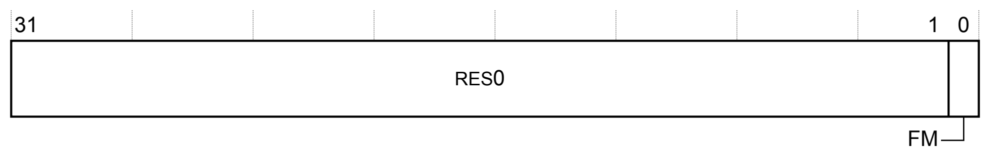


Table 2-15 FAULTMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, RES0
[0]	FM	In an implementation without the Security Extension, setting this bit to one boosts the current execution priority to -1, masking all exceptions except NMI. In an implementation 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 `0x0` **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. When the current execution priority is boosted to a particular value, all exceptions with a lower or equal priority are masked.

Base Priority Mask Register

Use the BASEPRI register to change the priority level that is required for exception preemption.

The bit assignments for the BASEPRI register are as follows:

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

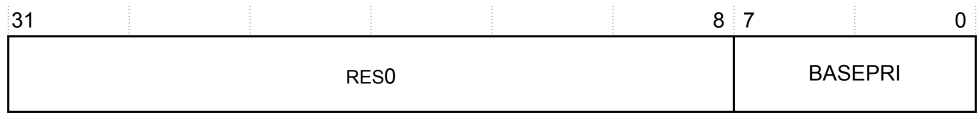


Table 2-16 BASEPRI register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, RES0
[7:0]	BASEPRI [g]	Software can boost the base priority by setting BASEPRI to a number between 1 and the maximum supported priority number. In an implementation with the Security Extension, the BASEPRI_NS is then mapped to the bottom half of the priority range, so that the current execution priority is boosted to the mapped value in the bottom half of the priority range. When the current execution priority is boosted to a particular value, all exceptions with a lower priority or equal preempting are masked. Writing 0 to BASEPRI disables base priority boosting.

CONTROL register

The CONTROL register controls the stack that is used, the privilege level for software execution when the core is in Thread mode and indicates whether the FPU or MVE state is active.

In an implementation with the Security Extension, this register is banked between Security states on a bit by bit basis.

The bit assignments for the CONTROL register when the Security Extension is implemented are as follows:

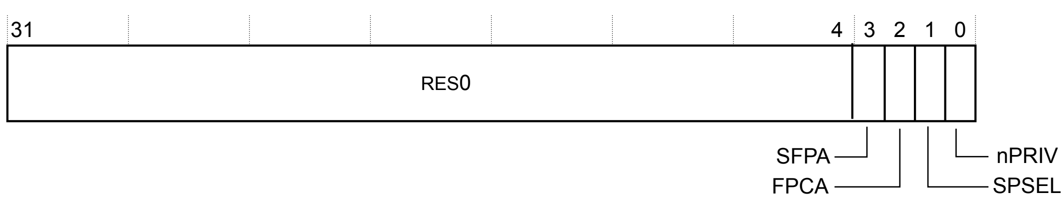


Figure 2-7 CONTROL bit assignments

Table 2-17 CONTROL register bit assignments

Bits	Name	Function
[31:4]	-	Reserved, RES0
[3]	SFPA	Indicates that the floating-point registers contain active state that belongs to the Secure state: **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 not banked between Security states and RAZ/WI from Non-secure state.

[g] This field is similar to the priority fields in the interrupt priority registers. If the device implements only bits[7:M] of this field, bits[M-1:0] read as zero and ignore writes. Higher priority field values correspond to lower exception priorities.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-17 CONTROL register bit assignments (continued)

Bits	Name	Function
[2]	FPCA	Indicates whether floating-point context is active: **0** No floating-point context active. **1** Floating-point context active. This bit is used to determine whether to preserve floating-point state when processing an exception. This bit is not banked between Security states.
[1]	SPSEL	Defines the currently active stack pointer: **0** MSP is the current stack pointer. **1** PSP is the current stack pointer. In Handler mode, this bit reads as zero and ignores writes. The Cortex-M55 core updates this bit automatically on exception return. This bit is banked between Security states.
[0]	nPRIV	Defines the Thread mode privilege level: **0** Privileged. **1** Unprivileged. This bit is banked between Security states.

The bit assignments for the CONTROL register when the Security Extension is not implemented are as follows:

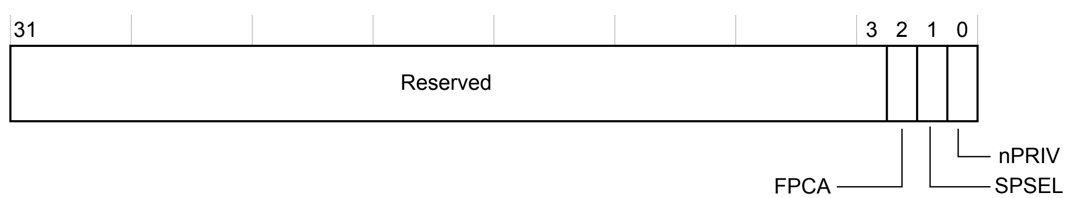


Figure 2-8 CONTROL bit assignments

Table 2-18 CONTROL register bit assignments

Bits	Name	Function
[31:3]	-	Reserved, RES0
[2]	FPCA	Indicates whether floating-point context is active: **0** No floating-point context active. **1** Floating-point context active. This bit is used to determine whether to preserve floating-point state when processing an exception.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-18 CONTROL register bit assignments (continued)

Bits	Name	Function
[1]	SPSEL	Defines the currently active stack pointer: **0** MSP is the current stack pointer. **1** PSP is the current stack pointer. In Handler mode, this bit reads as zero and ignores writes. The Cortex-M55 core updates this bit automatically on exception return.
[0]	nPRIV	Defines the Thread mode privilege level: **0** Privileged. **1** Unprivileged.

Handler mode always uses the MSP, so the processor ignores explicit writes to the active stack pointer bit of the CONTROL register when in Handler mode. The exception entry and return mechanisms automatically update the CONTROL register based on the EXC_RETURN value.

In an OS environment, Arm recommends that threads running in Thread mode use the process stack and the kernel and exception handlers use the main stack.

By default, Thread mode uses the MSP. To switch the stack pointer that is used in Thread mode to the PSP, either:
- Use the `MSR` instruction to set the CONTROL.SPSEL bit, the current active stack pointer bit, to 1.
- Perform an exception return to Thread mode with the appropriate EXC_RETURN value.

Note

When changing the stack pointer, software must use an `ISB` instruction immediately after the `MSR` instruction. This ensures that instructions after the `ISB` instruction execute using the new stack pointer.

Vector Predication Status and Control Register, VPR

The VPR holds the per-element predication flags.

This register is not banked between Security states.

In an implementation with the Security Extension, this register is not banked between Security states.

The bit assignments for the VPR register are as follows:

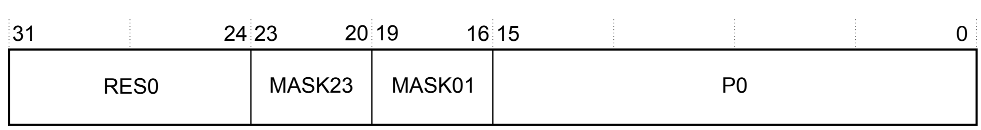


Figure 2-9 VPR bit assignments

The following table describes the VPR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-19 VPR bit assignments

Bits	Name	Function
[31:24]	-	Reserved, RES0
[23:20]	MASK23	The VPT mask bits for beat 2 and 3. `0b0000` Not in a VPT block. `0b1000` In a VPT block which is valid for one more instruction. The predicate flags are not inverted. `0bx100` In a VPT block which is valid for two more instructions. If set, the x bit causes the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block. `0bxx10` In a VPT block which is valid for three more instructions. If set, the x bits cause the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block. `0bxxx1` In a VPT block which is valid for four more instructions. If set, the x bits cause the predicate flags for beat 2 and 3 to be inverted between the corresponding instructions in the VPT block.
[19:16]	MASK01	The VPT mask bits for beat 0 and 1. `0b0000` Not in a VPT block. `0b1000` VPT predication valid for one more instruction. The predicate flags are not inverted. `0bx100` In a VPT block which is valid for two more instructions. If set, the x bit causes the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block. `0bxx10` In a VPT block which is valid for three more instructions. If set, the x bits cause the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block. `0bxxx1` In a VPT block which is valid for four more instructions. If set, the x bits cause the predicate flags for beat 0 and 1 to be inverted between the corresponding instructions in the VPT block.
[15:0]	P0	Predication bits. Each group of 4 bits determines the predication of each of the 4 bytes within the corresponding beat, regardless of instruction data type. **0** The corresponding vector lane will be masked. **1** The corresponding vector lane will be active.

2.1.4 Exceptions and interrupts

The Cortex-M55 processor implements all the logic required to handle and prioritize interrupts and other exceptions. Software can control this prioritization using the NVIC registers. All exceptions are vectored and, except for reset, are handled in Handler mode. Exceptions can target either Security state.

2.1.5 Data types and data memory accesses

The Cortex-M55 processor manages all data memory accesses as little-endian or big-endian. Instruction memory and *Private Peripheral Bus* (PPB) accesses are always performed as little-endian.

The processor supports the following data types:
- 32-bit words.
- 16-bit halfwords.
- 8-bit bytes.
- 16-bit half-precision floating-point numbers.
- 32-bit single-precision floating-point numbers.
- 64-bit double-precision floating-point numbers.

2.1.6 The Cortex Microcontroller Software Interface Standard

The *Cortex Microcontroller Software Interface Standard* (CMSIS) simplifies software development by enabling the reuse of template code and the combination of CMSIS-compliant software components

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

from various middleware vendors. Vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals.

For a Cortex-M55 microcontroller system, the CMSIS defines:

- A common way to:
  — Access peripheral registers.
  — Define exception vectors.
- The names of:
  — The registers of the core peripherals.
  — The core exception vectors.
- A device-independent interface for RTOS kernels, including a debug channel.
- Instruction primitives for instructions that do not map directly to `C`. For example, `LDAEX`. See *2.2.7 Synchronization primitives* on page 2-56

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M55 processor.

This document includes the register names defined by the CMSIS, and short descriptions of the CMSIS functions that address the processor core and the core peripherals.

Note

This document uses the register short names that are defined by the CMSIS. In a few cases these short names differ from the architectural short names that might be used in other documents.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2 Memory model

The Cortex-M55 processor has a fixed default memory map that provides up to 4GB of addressable memory.

2.2.1 Processor memory map

The default memory map for the Cortex-M55 processor covers the range `0x00000000-0xFFFFFFFF`.

Table 2-20 Default memory map

Address Range (inclusive)	Region	Interface
`0x00000000-0x1FFFFFFF`	Code	All accesses performed on the *Instruction Tightly Coupled Memory* (ITCM) or *Master-AXI* (M-AXI) interface.
`0x20000000-0x3FFFFFFF`	SRAM	All accesses performed on the *Data Tightly Coupled Memory* (DTCM) or M-AXI interface.
`0x40000000-0x5FFFFFFF`	Peripheral	• Data accesses performed on *Peripheral AHB* (P-AHB) or M-AXI interface. • Instruction accesses performed on M-AXI.
`0x60000000-0x9FFFFFFF`	External RAM	All accesses are performed on the M-AXI interface.
`0xA0000000-0xDFFFFFFF`	External device	All accesses are performed on the M-AXI interface.
`0xE0000000-0xE00FFFFF`	PPB	• Instruction fetches are not supported. • Reserved for system control and debug. • Data accesses are either performed internally or on *External Private Peripheral Bus* (EPPB).
`0xE0100000-0xFFFFFFFF`	Vendor_SYS	• Instruction fetches are not supported. • `0xE0100000-0xEFFFFFFF` reserved for future processor feature expansion. • Data accesses are performed on P-AHB interface.

The processor reserves regions of the *Private peripheral bus* (PPB) address range for core peripheral registers.

2.2.2 Memory regions, types, and attributes

If your implementation has an MPU or has the Security Extension MPUs, programming the relevant MPUs splits memory into regions.

The memory types are:

Normal
The processor can reorder transactions for efficiency, or perform Speculative reads.

Device
The processor preserves transaction order relative to other transactions to Device memory, and does not make speculative read accesses.

The additional memory attributes include:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Shareable

For a shareable memory region, the memory system might provide data synchronization between bus masters in a system with multiple bus masters, for example, a processor with a DMA controller.

If multiple bus masters can access a Non-shareable memory region, software must ensure data coherency between the bus masters.

Device memory is always Shareable.

Execute Never (XN)

This indicates that the processor prevents instruction accesses. A MemManage fault exception is generated on the attempt to execute the instruction whose fetch was prevented from an XN region of memory.

Cacheable

When a memory location is marked as Normal Cacheable, a copy of the memory location can be held in the cache, subject to aspects of the implementation.
Cacheable attributes can be Write-Through or Write-Back Cacheable.

Non-cacheable

When a memory location is marked as Non-cacheable, a copy of the memory location is not held in any cache.

2.2.3 Device memory

Device memory must be used for memory regions that cover peripheral control registers. Some of the optimizations that are permitted for Normal memory, such as access merging or repeating, can be unsafe for a peripheral register.

The Device memory type has several attributes:

G or nG Gathering or non-Gathering. Multiple accesses to a device can be merged into a single transaction except for operations with memory ordering semantics, for example, memory barrier instructions, load acquire/store release.

R or nR Reordering or non-Reordering.

E or nE Early Write Acknowledgement or no Early Write Acknowledgement. For the Cortex-M55 processor, nE Device transactions are buffered inside the processor itself. This attribute is then passed to the external interface to ensure that the response is received appropriately.

The Cortex-M55 processor treats all Device memory as nGnR. The only exception is *M-profile Vector Extension* (MVE) loads and stores in which multiple accesses from the same instruction might be gathered together.

The default memory map defines the Peripheral, External device, PPB, and Vendor_SYS regions as Device and the rest of the memory regions as Normal.

- Normal memory can be changed to Device.
- Device memory can be changed to Normal except for the following cases.
  - The PPB region is always Device-nGnRnE.
  - The Vendor_SYS region is Device-nGnRE and can be changed to Device-nGnRnE.
  - Mapping the Vendor_SYS region from Device to Normal results in UNPREDICTABLE behavior.

Typically, peripheral control registers must be either Device-nGnRE or Device-nGnRnE to prevent reordering of the transactions in the programming sequences.

2.2.4 Secure memory system and memory partitioning

In an implementation with the Security Extension, the *Security Attribution Unit* (SAU) and *Implementation Defined Attribution Unit* (IDAU) partition the 4GB memory space into Secure and Non-secure memory regions.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The partitioning of the memory into Secure and Non-secure regions is independent of the processor Security state. See *2.4 Security state switches* on page 2-76 for more information on Security state.

Secure memory partitioning

Secure addresses are used for memory and peripherals that are only accessible by Secure software or Secure masters. Transactions are deemed to be secure if they are to an address that is defined as Secure. Illegitimate accesses that are made by Non-secure software to Secure memory are blocked and raise an exception.

Non-secure Callable (NSC)

NSC is a special type of Secure location that is permitted to hold instructions starting with an `SG` instruction to enable software to transition from Non-secure to Secure state. The inclusion of NSC memory locations removes the need for Secure software creators to allow for the accidental inclusion of `SG` instructions, or data sharing the same encoding, in normal Secure memory by restricting the functionality of the `SG` instruction to NSC memory only.

Non-secure (NS)

Non-secure addresses are used for memory and peripherals accessible by all software running on the device.
Transactions are deemed to be Non-secure if they are to an address that is defined as Non-Secure.

Note

Transactions are deemed to be Non-secure even if secure software performs the access. Memory accesses initiated by Secure software to regions marked as Non-secure in the SAU and IDAU are marked as Non-secure on the external interfaces.

The MPU is banked between Secure and Non-secure memory. For instruction fetches, addresses that are Secure are subject to the Secure MPU settings. Addresses that are Non-secure are subject to the Non-secure MPU settings. For data loads and stores, accesses depend on the Security state of the processor.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.2.5 Behavior of memory accesses

The following table summarizes the behavior of accesses to each region in the default memory map.

Table 2-21 Memory access behavior

Address range	Memory region	Memory type	Shareability	XN	Description
`0x00000000-0x1FFFFFFF`	Code	Normal	Non-shareable	-	Executable region for program code. You can also put data here.
`0x20000000-0x3FFFFFFF`	SRAM	Normal	Non-shareable	-	Executable region for data. You can also put code here.
`0x40000000-0x5FFFFFFF`	Peripheral	Device, nGnRE	Shareable	XN	On-chip device memory.
`0x60000000-0x9FFFFFFF`	RAM	Normal	Non-shareable	-	Executable region for data.
`0xA0000000-0xDFFFFFFF`	External device	Device, nGnRE	Shareable	XN	External device memory.
`0xE0000000-0xE003FFFF`	Private Peripheral Bus	Device, nGnRnE	Shareable	XN	This region includes the SCS, NVIC, MPU, SAU, BPU, ITM, and DWT registers.
`0xE0040000-0xE0043FFF`	Device	Device, nGnRnE	Shareable	XN	This region is for debug components. Contact your implementer for more information.
`0xE0044000-0xE00FFFFF`	Private Peripheral Bus	Device, nGnRnE	Shareable	XN	This region includes the ROM tables.
`0xE0100000-0xFFFFFFFF`	Vendor_SYS	Device, nGnRE	Shareable	XN	Vendor specific.

Note

For more information on memory types, see *2.2.2 Memory regions, types, and attributes* on page 2-51.

The Code, SRAM, and RAM regions can hold programs.

The MPU can override the default memory access behavior described in this section.

Additional memory access constraints for caches and shared memory

When a system includes caches or shared memory, some regions of the default memory map have additional access constraints, and some regions are subdivided.

The following table shows the cache policies that are evident on the bus signal irrespective of the cache level.

Table 2-22 Memory region shareability and cache policies

Address range	Memory region	Memory type	Shareability	Cache policy
`0x00000000-0x1FFFFFFF`	Code	Normal	-	Write-Through
`0x20000000-0x3FFFFFFF`	SRAM	Normal	-	Write-Back Write-Allocate
`0x40000000-0x5FFFFFFF`	Peripheral	Device	Shareable	-
`0x60000000-0x7FFFFFFF`	RAM	Normal	-	Write-Back Write-Allocate
`0x80000000-0x9FFFFFFF`				Write-Through
`0xA0000000-0xDFFFFFFF`	External device	Device	Shareable	-

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-22 Memory region shareability and cache policies (continued)

Address range	Memory region	Memory type	Shareability	Cache policy
`0xE0000000-0xE00FFFFF`	Private Peripheral Bus	Device	Shareable	-
`0xE0100000-0xFFFFFFFF`	Vendor_SYS	Device	Shareable	Device

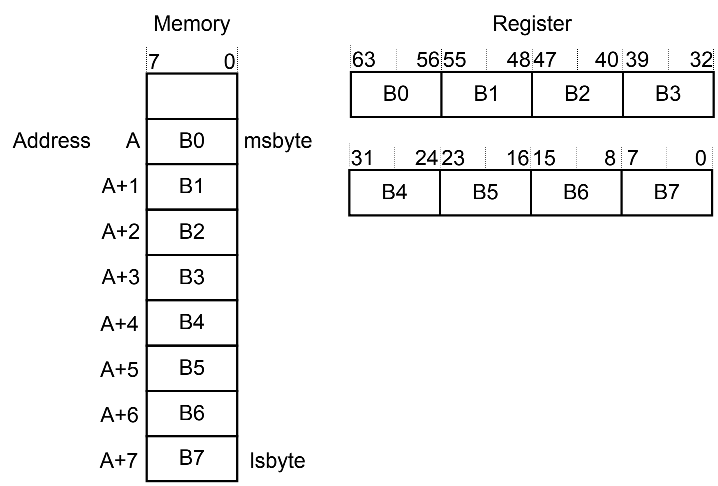
2.2.6 Memory endianness

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word.

Byte-invariant big-endian format

In byte-invariant big-endian format, the processor stores the *most significant byte* (msbyte) of a word at the lowest-numbered byte, and the *least significant byte* (lsbyte) at the highest-numbered byte.

Example 2-1 Byte-invariant big-endian example

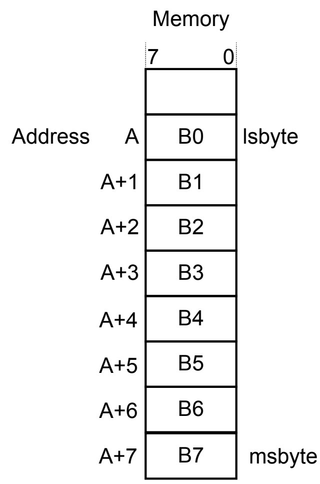


Little-endian format

In little-endian format, the processor stores the *least significant byte* (lsbyte) of a word at the lowest-numbered byte, and the *most significant byte* (msbyte) at the highest-numbered byte.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 2-2 Little-endian example



2.2.7 Synchronization primitives

The instruction set support for the processor includes pairs of *synchronization primitives*. These provide a non-blocking mechanism that a thread or process can use to obtain exclusive access to a memory location. Software can use them to implement semaphores or an exclusive read-modify-write memory sequence.

Instructions in synchronization primitives

A pair of synchronization primitives contains the following:

A Load-Exclusive instruction
Used to read the value of a memory location, requesting exclusive access to that location.

A Store-Exclusive instruction

Used to attempt to write to the same memory location, returning a status bit to a register. If this bit is:

0 It indicates that the thread or process gained exclusive access to the memory, and the write succeeded.

1 It indicates that the thread or process did not gain exclusive access to the memory, and no write was performed.

Load-Exclusive and Store-Exclusive instructions

The pairs of Load-Exclusive and Store-Exclusive instructions are:

- The word instructions:
  — `LDAEX` and `STLEX`.
  — `LDREX` and `STREX`.
- The halfword instructions:
  — `LDAEXH` and `STLEXH`.
  — `LDREXH` and `STREXH`.
- The byte instructions:
  — `LDAEXB` and `STLEXB`.
  — `LDREXB` and `STREXB`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Performing an exclusive read-modify-write

Software must use a Load-Exclusive instruction with the corresponding Store-Exclusive instruction.

To perform an exclusive read-modify-write of a memory location, the software must:
1. Use a Load-Exclusive instruction to read the value of the location.
2. Modify the value, as required.
3. Use a Store-Exclusive instruction to attempt to write the new value back to the memory location.
4. Test the returned status bit. If this bit is:

   0 The read-modify-write completed successfully.

   1 No write was performed. This indicates that the value returned at step 1 might be out of date. The software must retry the entire read-modify-write sequence.

Implementing a semaphore

The software can use the synchronization primitives to implement a semaphore as follows:
1. Use a Load-Exclusive instruction to read from the semaphore address to check whether the semaphore is free.
2. If the semaphore is free, use a Store-Exclusive to write the claim value to the semaphore address.
3. If the returned status bit from step 2 indicates that the Store-Exclusive succeeded, then the software has claimed the semaphore. However, if the Store-Exclusive failed, another process might have claimed the semaphore after the software performed step 1.

Exclusive tags

The processor includes an exclusive access monitor, that tags the fact that the processor has executed a Load-Exclusive instruction. If the processor is part of a multiprocessor system with a global monitor, and the address is in a shared region of memory, then the system also globally tags the memory locations that are addressed by exclusive accesses by each processor.

The processor clears its exclusive access tag if:

- It executes a `CLREX` instruction.
- It executes a `STREX` or `STLEX` instruction, regardless of whether the write succeeds.
- An exception occurs. This means that the processor can resolve semaphore conflicts between different threads.

In a multiprocessor implementation:
- Executing a `CLREX` instruction clears only the local exclusive access tag for the processor.
- Executing a `STREX` or `STLEX` instruction, or an exception, clears the local exclusive access tags for the processor.
- Executing a `STREX` or `STLEX` instruction to a Shareable memory region can also clear the global exclusive access tags for the processor and for any other master interface that has tagged the same address range in the system.

For more information about the synchronization primitive instructions, see *3.32.11 LDREX and STREX* on page 3-459 and *3.32.13 CLREX* on page 3-463.

For Normal memory, a global exclusive access can be performed in a Shared region if the MPU is implemented. In any other case, exclusive information is not sent on the AXI and AHB bus, and only the local monitor is used.

If **HEXCLP** or **AxLOCK** is sent externally on the P-AHB or M-AXI bus respectively, and there is no exclusive monitor for the corresponding memory region, then exclusive stores fail.

2.2.8 Programming hints for the synchronization primitives

ISO/IEC C cannot directly generate the exclusive access instructions. CMSIS provides intrinsic functions for generation of these instructions.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-23 CMSIS functions for exclusive access instructions

Instruction	CMSIS function
LDAEX	uint32_t __LDAEX (volatile uint32_t * ptr)
LDAEXB	uint8_t __LDAEXB (volatile uint8_t * ptr)
LDAEXH	uint16_t __LDAEXH (volatile uint16_t * ptr)
LDREX	uint32_t __LDREXW (uint32_t *addr)
LDREXB	uint8_t __LDREXB (uint8_t *addr)
LDREXH	uint16_t __LDREXH (uint16_t *addr)
STLEX	uint32_t __STLEX (uint32_t value, volatile uint32_t * ptr)
STLEXB	uint8_t __STLEXB (uint8_t value, volatile uint8_t * ptr)
STLEXH	uint16_t __STLEXH (uint16_t value, volatile uint16_t * ptr)
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXB	uint8_t __STREXB (uint8_t value, uint8_t *addr)
STREXH	uint16_t __STREXH (uint16_t value, uint16_t *addr)
CLREX	void __CLREX (void)

For example:

```
uint16_t  value;
uint16_t *address = 0x20001002;
value = __LDREXH (address);    // load 16-bit value from memory address 0x20001002
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.3 Exception model

This section contains information about different parts of the exception model such as exception types, exception priorities, and exception states.

2.3.1 Exception states

An exception can be in any of the following states:

Inactive

The exception is not active and not pending.

Pending

The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

Active

An exception is being serviced by the processor but has not completed. An exception handler can interrupt the execution of another exception handler. In this case, both exceptions are in the active state.

Active and pending

The exception is being serviced by the processor and there is a pending exception from the same source.

2.3.2 Exception types

This section describes the exception types of the processor.

Exception types with the Security Extension

Reset

The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When either power-on or warm reset is deasserted, execution restarts from the address provided by the reset entry in the Secure vector table. Execution restarts as privileged execution in Secure state in Thread mode.

This exception is not banked between Security states.

NMI

A *Non-Maskable Interrupt* (NMI) can be signaled by a peripheral or triggered by software. It is permanently enabled and has a fixed priority of -2. NMI can only be preempted by reset and, when it is Non-secure, by a Secure HardFault.

If AIRCR.BFHFNMINS=0, then the NMI is Secure.

If AIRCR.BFHFNMINS=1, then NMI is Non-secure.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HardFault

A HardFault is an exception that occurs because of an error during normal or exception processing. HardFaults have a fixed priority of at least -1, meaning they have higher priority than any exception with configurable priority.

This exception is not banked between Security states.

If AIRCR.BFHFNMINS=0, HardFault handles all faults that are unable to preempt the current execution. The HardFault handler is always Secure.

If AIRCR.BFHFNMINS=1, HardFault handles faults that target Non-secure state that are unable to preempt the current execution.

HardFaults that specifically target the Secure state when AIRCR.BFHFNMINS is set to 1 have a priority of -3 to ensure they can preempt any execution. A Secure HardFault at Priority -3 is only enabled when AIRCR.BFHFNMINS is set to 1. Secure HardFault handles Secure faults that are unable to preempt current execution.

MemManage

A MemManage fault is an exception that occurs because of a memory protection violation, compared to the MPU or the fixed memory protection constraints, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to *Execute Never* (XN) memory regions.

This exception is banked between Security states.

BusFault

A BusFault is an exception that occurs because of a memory-related violation for an instruction or data memory transaction. This might be from an error that is detected on a bus in the memory system.

This exception is not banked between Security states.

If BFHFNMINS=0, BusFaults target the Secure state.

If BFHFNMINS=1, BusFaults target the Non-secure state.

UsageFault

A UsageFault is an exception that occurs because of a fault related to instruction execution. This includes:
- An undefined instruction.
- An illegal unaligned access.
- Invalid state on instruction execution.
- An error on exception return.

The following can cause a UsageFault when the core is configured by software to report them:
- An unaligned address on word and halfword memory access.
- Division by zero.

This exception is banked between Security states.

SecureFault

This exception is triggered by the various security checks that are performed. It is triggered, for example, when jumping from Non-secure code to an address in Secure code that is not marked as a valid entry point. Most systems choose to treat a SecureFault as a terminal condition that either halts or restarts the system. Any other handling of the SecureFault must be checked carefully to make sure that it does not inadvertently introduce a security vulnerability. SecureFaults always target the Secure state.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SVCall

A *Supervisor Call* (SVC) is an exception that is triggered by the `SVC` instruction. In an OS environment, applications can use `SVC` instructions to access OS kernel functions and device drivers.

This exception is banked between Security states.

DebugMonitor

A DebugMonitor exception. If Halting debug is disabled and the debug monitor is enabled, a debug event causes a DebugMonitor exception when the group priority of the DebugMonitor exception is greater than the current execution priority. Debug monitor does support Secure and Non-secure state. The target Security state is determined by DEMCR.SDME, Secure Debug Monitor Enable, which is a reflection of the debug authentication state.

- If this field is 0, DebugMonitor events are only allowed in Non-secure and the exception uses the Non-secure vector table and targets the Non-secure DebugMonitor handler
- If this field is 1, DebugMonitor events are allowed in Secure and Non-secure state and the exception uses the Secure vector table and targets the Secure DebugMonitor handler

DEMCR.SDME is read only and reflects of the self-hosted debug authentication state based on DAUTHCTRL, DBGEN, SPIDEN. The register field cannot be changed when a DebugMonitor exception is pending or active.

PendSV

PendSV is an asynchronous request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

This exception is banked between Security states.

SysTick

A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as a system tick.

This exception is banked between Security states.

Interrupt (IRQ)

An interrupt, or IRQ, is an exception signaled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

This exception is not banked between Security states. Secure code can assign each interrupt to Secure or Non-secure state. By default all interrupts are assigned to Secure state.

Privileged software can disable the exceptions that have configurable priority, as shown in the following table.

Table 2-24 Properties of the different exception types with the Security Extension

Exception number (see notes)	IRQ number (see notes)	Exception type	Priority	Vector address	Activation
1	-	Reset	-4, the highest	`0x00000004`	Asynchronous
2	-14	NMI	-2	`0x00000008`	Asynchronous

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-24 Properties of the different exception types with the Security Extension (continued)

Exception number (see notes)	IRQ number (see notes)	Exception type	Priority	Vector address	Activation
3	-13	Secure HardFault when AIRCR.BFHFNMINS is 1	-3	`0x0000000C`	Synchronous
		Secure HardFault when AIRCR.BFHFNMINS is 0	-1		
		HardFault	-1		
4	-12	MemManage	Configurable	`0x00000010`	Synchronous
5	-11	BusFault	Configurable	`0x00000014`	Synchronous
6	-10	UsageFault	Configurable	`0x00000018`	Synchronous
7	-9	SecureFault	Configurable	`0x0000001C`	Synchronous
8-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable	`0x0000002C`	Synchronous
12	-4	DebugMonitor	Configurable	`0x00000030`	Synchronous
13	-	Reserved	-	-	-
14	-2	PendSV	Configurable	`0x00000038`	Asynchronous
15	-1	SysTick	Configurable	`0x0000003C`	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable	`0x00000040` and above. Increasing in steps of 4	Asynchronous

Note

- To simplify the software layer, the CMSIS only uses IRQ numbers. It uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-40.
- For configurable priority values, see *4.2.7 Interrupt Priority Registers* on page 4-470.

For an asynchronous exception, other than reset, the processor can execute extra instructions between the moment the exception is triggered and the moment the processor enters the exception handler.

An exception that targets Secure state cannot be disabled by Non-secure code.

Exception types without the Security Extension

Reset

The exception model treats reset as a special form of exception. When either power-on or warm reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.

NMI

A *Non-Maskable Interrupt* (NMI) can be signaled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be masked or preempted by any exception other than Reset.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

HardFault

A HardFault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. HardFaults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.

MemManage

A MemManage fault is an exception that occurs because of a memory protection violation, compared to the MPU or the fixed memory protection constraints, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to *Execute Never* (XN) memory regions.

BusFault

A BusFault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction. This might be from an error that is detected on a bus in the memory system.

UsageFault

A UsageFault is an exception that occurs because of a fault related to instruction execution. This includes:

- An undefined instruction.
- An illegal unaligned access.
- Invalid state on instruction execution.
- An error on exception return.

The following can cause a UsageFault when the core is configured by software to report them:

- An unaligned address on word and halfword memory access.
- Division by zero.

SVCall

A *Supervisor Call* (SVC) is an exception that is triggered by the `SVC` instruction. In an OS environment, applications can use `SVC` instructions to access OS kernel functions and device drivers.

DebugMonitor

A DebugMonitor exception. If Halting debug is disabled and the debug monitor is enabled, a debug event causes a DebugMonitor exception when the group priority of the DebugMonitor exception is greater than the current execution priority.

PendSV

PendSV is an asynchronous request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

SysTick

A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as a system tick.

Interrupt (IRQ)

An interrupt, or IRQ, is an exception signaled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Privileged software can disable the exceptions that have configurable priority, as shown in the following table.

Table 2-25 Properties of the different exception type without the Security Extensions

Exception number (see notes)	IRQ number (see notes)	Exception type	Priority	Vector address	Activation
1	-	Reset	-4, the highest	`0x00000004`	Asynchronous
2	-14	NMI	-2	`0x00000008`	Asynchronous
3	-13	HardFault	-1	`0x0000000C`	Synchronous
4	-12	MemManage	Configurable	`0x00000010`	Synchronous
5	-11	BusFault	Configurable	`0x00000014`	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable	`0x00000018`	Synchronous
7-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable	`0x0000002C`	Synchronous
12	-4	DebugMonitor	Configurable	`0x00000030`	Synchronous
13	-	Reserved	-	-	-
14	-2	PendSV	Configurable	`0x00000038`	Asynchronous
15	-1	SysTick	Configurable	`0x0000003C`	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable	`0x00000040` and above. Increasing in steps of 4	Asynchronous

Note

- To simplify the software layer, the CMSIS only uses IRQ numbers. It uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-40.
- For configurable priority values, see *4.2.7 Interrupt Priority Registers* on page 4-470.

For an asynchronous exception, other than reset, the processor can execute extra instructions between the moment the exception is triggered and the moment the processor enters the exception handler.

2.3.3 Exception handlers

Exception handlers are designed to be standard C/C++ functions that are entered using standard calling conventions. If handlers are written in assembly code, then standard function calling conventions that are described in the Procedure Call standard for Arm architecture must be followed.

Interrupt Service Routines (ISRs)

Interrupts IRQ0-IRQ479 are the exceptions that are handled by ISRs.

In an implementation with the Security Extension, each interrupt is configured by Secure software to execute in Secure or Non-secure state, using NVIC_ITNS.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Fault handlers

The fault handlers handle the following exceptions:

- HardFault.
- MemManage.
- BusFault.
- UsageFault.
- SecureFault, when the Security Extension is implemented.

In an implementation with the Security Extension, there can be separate MemManage and UsageFault handlers in Secure and Non-secure state. The AIRCR.BFHFNMINS bit controls the target state for HardFault and BusFault. SecureFault always targets Secure State.

System handlers

The system handlers handle the following system exceptions:

- NMI.
- PendSV.
- SVCall.
- SysTick.

In an implementation with the Security Extension, most system handlers can be banked with separate handlers between Secure and Non-secure state. The AIRCR.BFHFNMINS bit controls the target state for NMI.

2.3.4 Vector table

The VTOR register in the *System Control Block* (SCB) determines the starting address of the vector table. In an implementation with the Security Extension, the VTOR is banked so there is a VTOR_S and a VTOR_NS. The initial values of VTOR_S and VTOR_NS are system design specific. The vector table used depends on the target state of the exception. For exceptions targeting the Secure state, VTOR_S is used. For exceptions targeting the Non-secure state, VTOR_NS is used.

Vector table without the Security Extension

The following figure shows the order of the exception vectors in the vector table for an implementation without the Security Extension. The least-significant bit of each vector is 1, indicating that the exception handler is written in Thumb code.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception number	IRQ number	Vector	Offset
495	479	IRQ479	0x7BC
.		.	.
.		.	.
.		.	.
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick	0x3C
14	-2	PendSV	0x38
13		Reserved	0x30
12	-4	DebugMonitor	
11	-5	SVCall	0x2C
10			
9		Reserved	
8			
7	-9	SecureFault	0x1C
6	-10	UsageFault	0x18
5	-11	BusFaults	0x14
4	-12	MemManage	0x10
3	-13	HardFault	0x0C
2	-14	NMI	0x08
1		Reset	0x04
		Initial SP value	0x00

Figure 2-10 Vector table without the Security Extension

On system reset the vector table is set to the value of the external **INITNSVTOR** bus. Privileged software can write to VTOR to relocate the vector table start address to a different memory location, in the range `0x00000000` to `0xFFFFFF80`, assuming access is allowed by the external **LOCKNSVTOR** pin.

The silicon vendor must configure the required alignment, which depends on the number of interrupts implemented. The minimum alignment is 32 words, enough for up to 16 interrupts. For more interrupts, adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, the alignment must be on a 64-word boundary because the required table size is 37 words, and the next power of two is 64.

Vector table with the Security Extension

The following figure shows the order of the exception vectors in the Secure and Non-secure vector tables. The least-significant bit of each vector is 1, indicating that the exception handler is written in Thumb code.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Exception number	IRQ number	Vector	Offset
495	479	IRQ479	0x7BC
. . .		. . .	. . .
18	2	IRQ2	0x48
17	1	IRQ1	0x44
16	0	IRQ0	0x40
15	-1	SysTick	0x3C
14	-2	PendSV	0x38
13		Reserved	0x30
12	-4	DebugMonitor	
11	-5	SVCall	0x2C
10		Reserved	
9			
8			
7	-9	SecureFault	0x1C
6	-10	UsageFault	0x18
5	-11	BusFaults	0x14
4	-12	MemManage	0x10
3	-13	HardFault	0x0C
2	-14	NMI	0x08
1		Reset	0x04
		Initial SP value	0x00

Figure 2-11 Vector table with the Security Extension

Because reset always targets Secure state, the Non-secure Reset and Non-secure initial SP value are ignored by the hardware. However, from a software perspective, Secure boot code might provide the initial SP value during Non-secure initialization.

On system reset, the Non-secure vector table is set to the value of the external **INITNSVTOR** bus, and the Secure vector table is set to the value of the external **INITSVTOR** bus. Privileged software can write to VTOR_S and VTOR_NS to relocate the vector table start address to a different memory location, in the range `0x00000000` to `0xFFFFFF80`, assuming access is allowed by the external **LOCKNSVTOR** and **LOCKSVTAIRCR** pins respectively.

The silicon vendor must configure the required alignment of the vector tables, which depends on the number of interrupts implemented. The minimum alignment is 32 words, enough for up to 16 interrupts. For more interrupts, adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, the alignment must be on a 64-word boundary because the required table size is 37 words, and the next power of two is 64.

2.3.5 Exception priorities

All exceptions have an assigned priority value that is used to control both preemption and prioritization between pending exceptions. A lower priority value indicates a higher priority. You can configure priority values for all exceptions except Reset, HardFault, and NMI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If software does not configure any priority values, then all exceptions with a configurable priority value have a priority of 0. For information about configuring exception priorities, see:
- *4.3.33 System Handler Priority Registers* on page 4-524.
- *4.2.7 Interrupt Priority Registers* on page 4-470.

Note

Configurable priority values are in the range 0-255. The Reset, HardFault, and NMI exceptions, with fixed negative priority values always have higher priority than any other exception.

If the Security Extension is implemented, for configurable priority values, the target Security state also affects the programmed priority. Depending on the value of AIRCR.PRIS, the priority can be extended.

In the table, the values in columns 2 and 3 must match, and increase from zero in increments of 32. The values in column 4 start from 128 and increase in increments of 16.

The following table enumerates the effective priority values that are available when a 3-bit priority is configured for the processor. This results in eight possible levels of priority.

Table 2-26 Extended priority

Priority value [7:5]	Secure priority	Non-secure priority when AIRCR.PRIS = 0	Non-secure priority when AIRCR.PRIS = 1
0	0	0	128
1	32	32	144
2	64	64	160
3	96	96	176
4	128	128	192
5	160	160	208
6	192	192	224
7	224	224	240

Assigning a higher priority value to IRQ[0] and a lower priority value to IRQ[1] means that IRQ[1] has higher priority than IRQ[0]. If both IRQ[1] and IRQ[0] are asserted, IRQ[1] is processed before IRQ[0].

If multiple pending exceptions have the same priority, the pending exception with the lowest exception number takes precedence. For example, if both IRQ[0] and IRQ[1] are pending and have the same priority, then IRQ[0] is processed before IRQ[1].

When the processor is executing an exception handler, the exception handler is preempted if a higher priority exception occurs. If an exception occurs with the same priority as the exception being handled, the handler is not preempted, irrespective of the exception number. However, the status of the new interrupt changes to pending.

2.3.6 Interrupt priority grouping

To increase priority control in systems with interrupts, the NVIC supports priority grouping. This divides each interrupt priority register entry into two fields, an upper field that defines the *group priority*, and a lower field that defines a *subpriority* within the group.

Only the group priority determines pre-emption of interrupt exceptions. When the processor is executing an interrupt exception handler, another interrupt with the same group priority as the interrupt being handled does not pre-empt the handler.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If multiple pending interrupts have the same group priority, the subpriority field determines the order in which they are processed. If multiple pending interrupts have the same group priority and subpriority, the interrupt with the lowest IRQ number is processed first.

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.

If the implementation include the Security Extension, then, 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.

2.3.7 Exception handling

The exception handling mechanism includes preemption, exception return, tail-chaining, and handling late-arriving interrupts.

Preemption

An exception can preempt the current execution if its priority is higher than the current execution priority.

When one exception preempts another, the exceptions are called nested exceptions.

Return

This occurs when the exception handler is completed.

The processor pops the stack and restores the processor state to the state it had before the interrupt occurred.

Tail-chaining

This mechanism speeds up exception servicing. On completion of an exception handler or during the return operation, if there is a pending exception that meets the requirements for exception entry, then the stack pop is skipped and control transfers directly to the new exception handler.

Late arriving interrupts

This mechanism speeds up preemption. If a higher priority exception occurs during state saving for a previous exception, the processor switches to handle the higher priority exception and initiates the vector fetch for that exception. State saving may be affected by the late arrival depending on the stacking requirements of the original exception and the late-arriving exception. On return from the exception handler of the late-arriving exception, the normal tail-chaining rules apply.

Exception entry

Exception entry occurs when there is a pending exception with sufficient priority and either the processor is in Thread mode, or the new exception is of higher priority than the exception being handled, in which case the new exception preempts the original exception.

When one exception preempts another, the exceptions are nested.

Sufficient priority means that the exception has higher priority than any limits set by the mask registers. An exception with lower priority than this is pending but is not handled by the processor.

When the processor takes an exception, unless the exception is a tail-chained or a late-arriving exception, the processor pushes information onto the current stack. This operation is referred to as *stacking* and the structure of the data stacked is referred as the *stack frame*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If the floating-point context is active, the Cortex-M55 processor can automatically stack the architected floating-point state on exception entry. The following figure shows the Cortex-M55 processor stack frame layout when an interrupt or an exception is preserved on the stack:

- With floating-point state.
- Without floating-point state.

Note

Where stack space for floating-point state is not allocated, the stack frame is the same as that of Armv8.1-M implementations without floating-point operation.

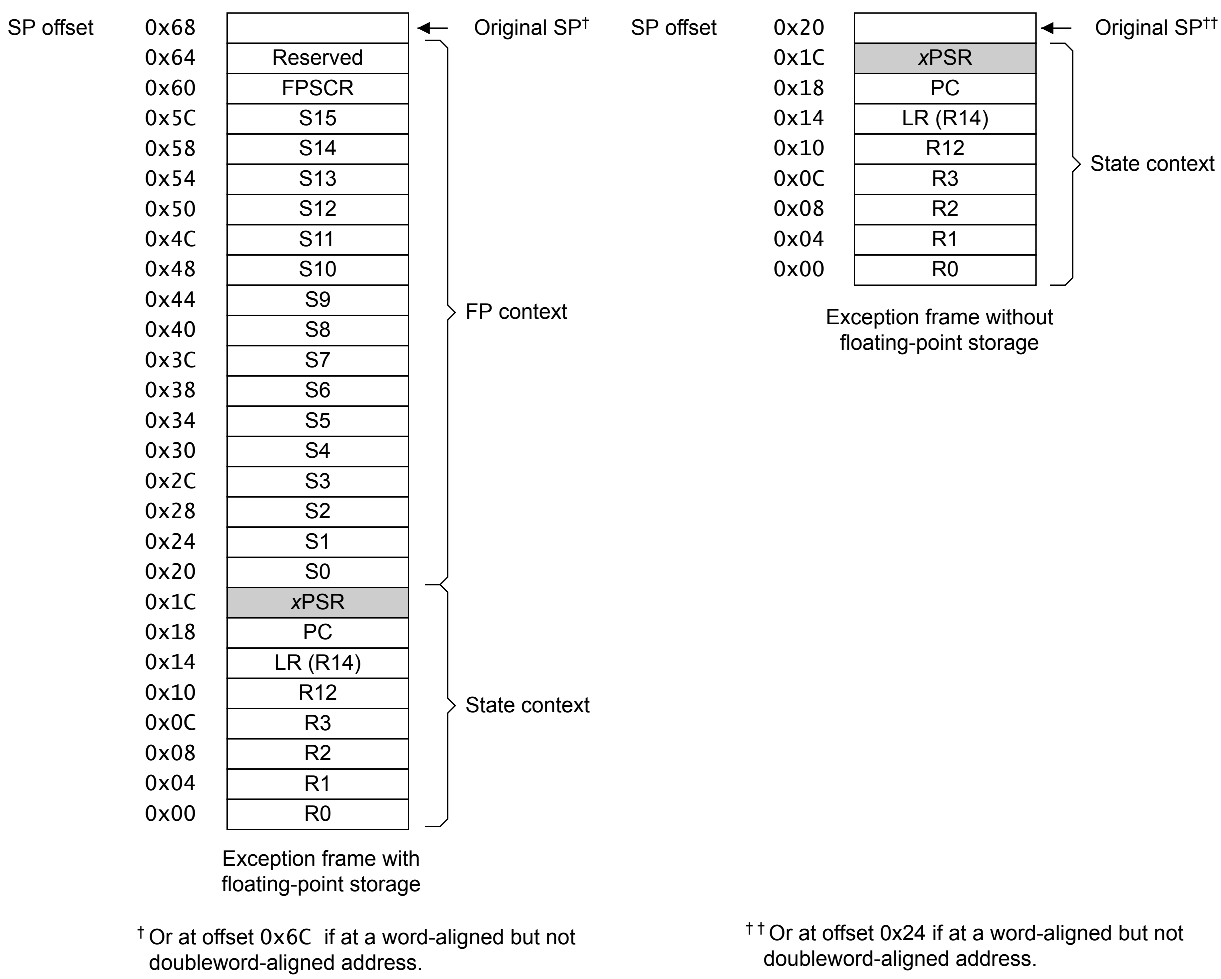


Figure 2-12 Stack frame when an interrupt or an exception is preserved on the stack with or without floating-point state

If the Security Extension is implemented, when a Non-secure exception preempts software running in a Secure state, additional context is saved onto the stack and the stacked registers are cleared to ensure no Secure data is available to Non-secure software, as the following figure shows.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

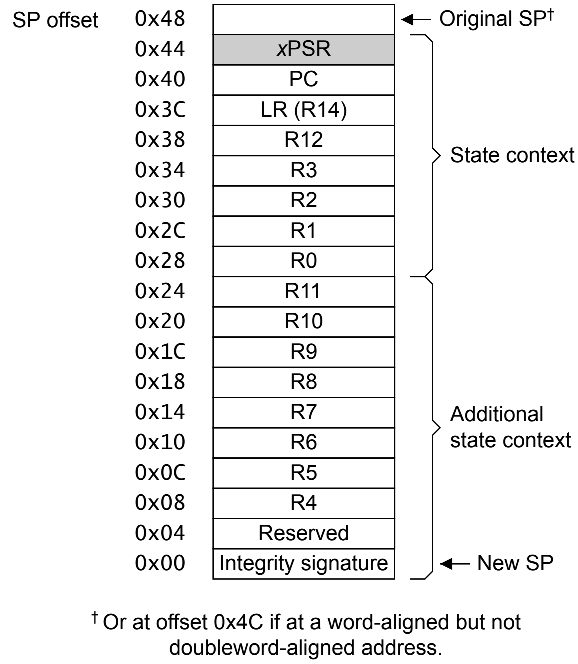


Figure 2-13 Stack frame extended to save additional context when the Security Extension is implemented

If the floating-point context is active, the Cortex-M55 processor automatically stacks floating-point state in the stack frame. There are two frame formats that contain floating-point context. If an exception is taken from Secure state and FPCCR.TS is set, the additional floating-point context is stacked. In all other cases, only the standard floating-point context is stacked, as the following figure shows.

Note

The conditions that trigger saving additional FP context are different from those that trigger additional integer context.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

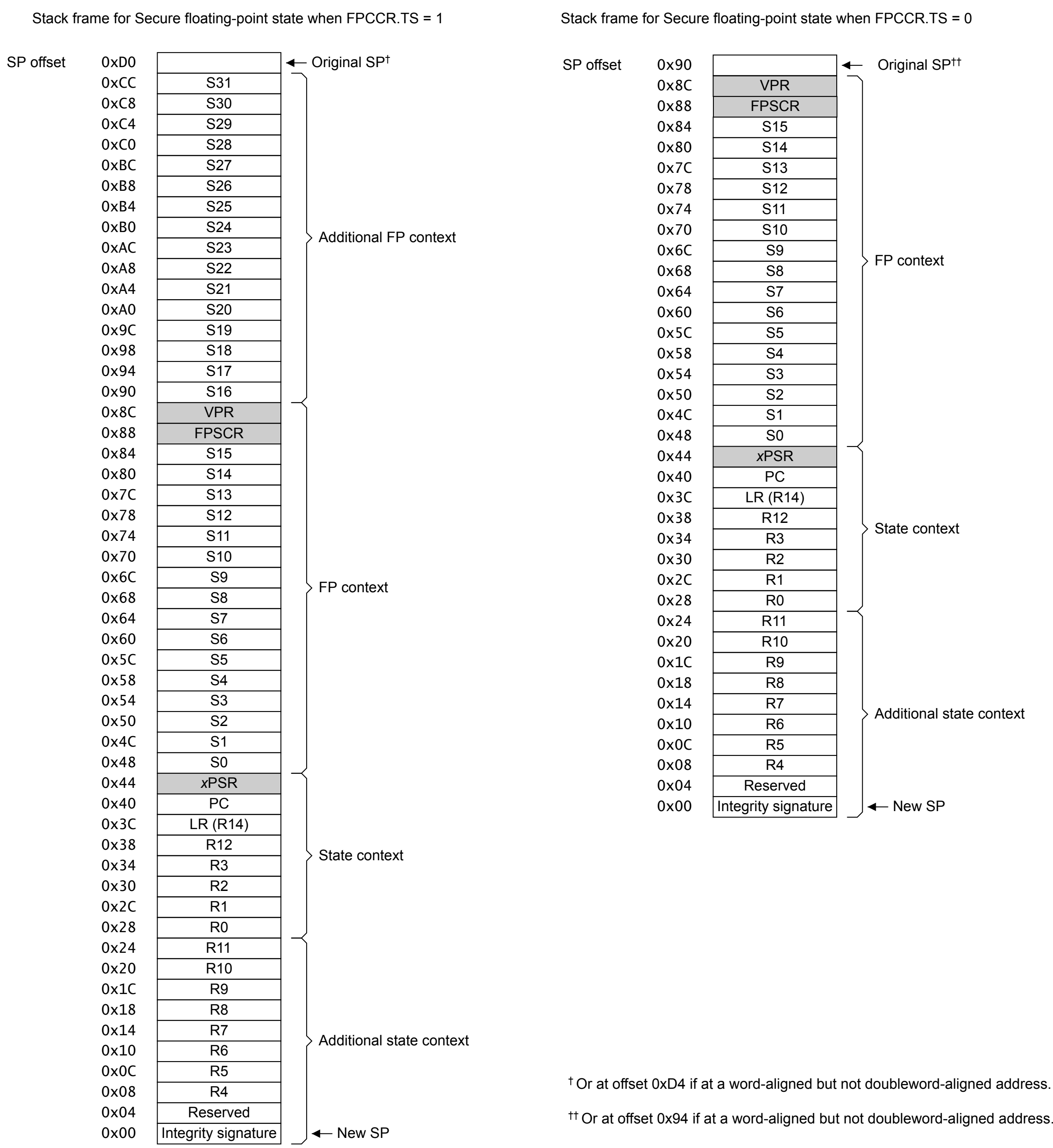


Figure 2-14 Extended exception stack frame

The Stack pointer of the interrupted thread or handler is always used for stacking the state before the exception is taken. For example if an exception is taken from Secure state to a Non-secure handler the Secure stack pointer is used to save the state.

Immediately after stacking, the stack pointer indicates the lowest address in the stack frame.

The stack frame includes the return address. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

In parallel to the stacking operation, the processor performs a vector fetch that reads the exception handler start address from the vector table. When stacking is complete, the processor starts executing the

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

exception handler. At the same time, the processor writes an EXC_RETURN value to the LR. This value is used to trigger exception return when the exception handler is complete.

If no higher priority exception occurs during exception entry, the processor starts executing the exception handler and automatically changes the status of the corresponding pending interrupt to active.

If another higher priority exception occurs during exception entry, the processor starts executing the exception handler for this exception and does not change the pending status of the earlier exception. This is the late arrival case.

Exception return

Exception return occurs when the processor is in Handler mode and execution of one of the following instructions attempts to set the PC to an EXC_RETURN value.

- A `POP` or `LDM` instruction that loads the PC.
- An `LDR` instruction that loads the PC.
- A `BX` instruction using any register.

Exception return in an implementation with the Security Extension

The processor saves an EXC_RETURN value to the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. When the processor loads a value matching this pattern to the PC it detects that the operation is not a normal branch operation and, instead, that the exception is complete. As a result, it starts the exception return sequence. Bits[6:0] of the EXC_RETURN value indicate the required return stack, processor mode, Security state, and stack frame as the following table shows.

Table 2-27 Exception return behavior, EXC_RETURN bit field description

Bits	Name	Function
[31:24]	PREFIX	Prefix. Indicates that this is an EXC_RETURN value. This field reads as `0b11111111`.
[23:7]	-	Reserved, RES1.
[6]	S	Secure or Non-secure stack. Indicates whether registers have been pushed to a Secure or Non-secure stack. **0** Non-secure stack used. **1** Secure stack used.
[5]	DCRS	Default callee register stacking. Indicates whether the default stacking rules apply, or whether the callee registers are already on the stack. **0** Stacking of the callee saved registers is skipped. **1** Default rules for stacking the callee registers are followed.
[4]	FType	In a PE with the Main and Floating-point Extensions: **0** The PE allocated space on the stack for FP context. **1** The PE did not allocate space on the stack for FP context. In a PE without the Floating-point Extension, this bit is Reserved, RES1.
[3]	Mode	Indicates the mode that was stacked from. **0** Handler mode. **1** Thread mode.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-27 Exception return behavior, EXC_RETURN bit field description (continued)

Bits	Name	Function
[2]	SPSEL	Indicates which stack contains the exception stack frame. **0** Main stack pointer. **1** Process stack pointer.
[1]	-	Reserved, RES0.
[0]	ES	Indicates the Security state the exception was taken to. **0** Non-secure. **1** Secure.

Exception return in an implementation without the Security Extension

The processor saves an EXC_RETURN value to the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. When the processor loads a value matching this pattern to the PC it detects that the operation is not a normal branch operation and, instead, that the exception is complete. As a result, it starts the exception return sequence. Bits[6:0] of the EXC_RETURN value indicate the required return stack, processor mode, and stack frame as the following table shows.

Table 2-28 Exception return behavior, EXC_RETURN bit field description

Bits	Name	Function
[31:24]	PREFIX	Prefix. Indicates that this is an EXC_RETURN value. This field reads as `0b11111111`.
[23:7]	-	Reserved, RES1.
[6]	-	Reserved, RES0.
[5]	DCRS	Default callee register stacking. Indicates whether the default stacking rules apply, or whether the callee registers are already on the stack. **0** Stacking of the callee saved registers is skipped. **1** Default rules for stacking the callee registers are followed.
[4]	FType	Stack frame type. In a PE with the standard integer and Floating-point Extensions: **0** The PE allocated space on the stack for FP context. **1** The PE did not allocate space on the stack for FP context. Stack frame type. In a PE without the Floating-point Extension, this bit is Reserved, RES1.
[3]	Mode	Indicates the mode that was stacked from. **0** Handler mode. **1** Thread mode.
[2]	SPSEL	Stack pointer selection. Indicates which stack contains the exception stack frame. **0** Main stack pointer. **1** Process stack pointer.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-28 Exception return behavior, EXC_RETURN bit field description (continued)

Bits	Name	Function
[1]	-	Reserved, RES0.
[0]	-	UNPREDICTABLE

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.4 Security state switches

The following table presents the possible security transitions, the instructions that can cause them, and any faults that may be generated.

Table 2-29 Security state transitions

Current Security state	Security attribute of the branch target address	Security state change
Secure	Non-secure	Change to Non-secure state if the branch was a `BXNS` or `BLXNS` instruction, with the lsb of its target address set to 0. Otherwise, a SecureFault is generated.
Non-secure	Secure and Non-secure callable	Change to Secure state if the branch target address contains an `SG` instruction. If the target address does not contain an SG a SecureFault is generated.
Non-secure	Secure and not Non-secure callable	A SecureFault is generated.

The following figure shows the Security state transitions:

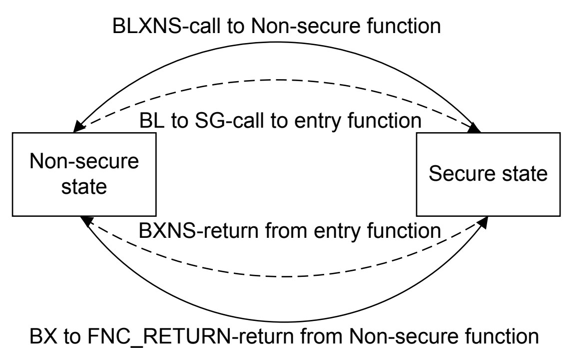


Figure 2-15 Security state transitions

Secure software can call a Non-secure function using the `BLXNS` instruction. When this happens, the LR is set to a special value called FNC_RETURN, and the return address and XPSR is saved onto the Secure stack. Return from Non-secure state to Secure state is triggered when one of the following instructions attempts to set the PC to an FNC_RETURN value:

- A `POP` or `LDM` instruction that loads the PC.
- An `LDR` instruction that loads the PC.
- A `BX` instruction using any register.

When a return from Non-secure state to Secure state occurs the processor restores the program counter and XPSR from the Secure stack.

Any scenario not listed in the table triggers a SecureFault. For example, sequential instructions that cross security attributes from Secure to Non-secure or from Non-secure to Secure.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.5 Fault handling

Faults can occur on instruction fetches, instruction execution, and data accesses. When a fault occurs, information about the cause of the fault is recorded in various registers, according to the type of fault. Faults are a subset of the exceptions.

Faults occur on instructions fetches for the following reasons:

- *Memory Protection Unit* (MPU) MemManage fault.
- *Security Attribution Unit* (SAU) or *Implementation Defined Attribution Unit* (IDAU) SecureFault.
- BusFaults that are caused by an external AXI slave error (SLVERR), and external AXI decode error (DECERR), or corrupted transactions (RPOISON).
- BusFaults caused by TCM external errors.
- BusFaults caused by uncorrectable *Error Correcting Code* (ECC) errors in the TCM.
- BusFault caused by a poisoned TEBRx register.
- BusFaults caused by *TCM Gate Unit* (TGU) faults.

Faults can occur on data accesses for the following reasons:

- MPU MemManage fault.
- Alignment UsageFault.
- SAU or IDAU SecureFault.
- BusFaults that are caused by an external AXI slave error (SLVERR) , an external AXI decode error (DECERR), or corrupted read data (RPOISON).
- BusFaults because of errors on the *External Private Peripheral Bus* (EPPB) APB interface.
- External AHB error from the *Peripheral-AHB* (P-AHB) interface.
- BusFaults caused by TCM external errors.
- BusFaults caused by uncorrectable *Error Correcting Code* (ECC) errors in the TCM or *Level 1* (L1) data cache.
- BusFault caused by a poisoned TEBRx register.
- *M-profile Vector Extension* (MVE) transactions, stacking, or unstacking to the PPB space.
- BusFaults caused by TGU faults.
- Unprivileged accesses to system registers which only privileged code can access.

2.5.1 Fault types reference table

The table shows the types of fault, the handler used for the fault, the corresponding fault status register, and the register bit that indicates that the fault has occurred.

Table 2-30 Fault types and fault status registers

Fault	Handler	Bit name	Fault status register
Bus error on a vector read	HardFault	VECTTBL	*4.3.17 HardFault Status Register* on page 4-503
Fault escalated to a hard fault		FORCED	

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 2-30 Fault types and fault status registers (continued)

Fault	Handler	Bit name	Fault status register
MPU or default memory map mismatch on instruction access	MemManage	IACCVIOL [h]	*MemManage Fault Status Register* on page 4-496
MPU or default memory map mismatch on data access		DACCVIOL	
MPU or default memory map mismatch during exception stacking		MSTKERR	
MPU or default memory map mismatch during exception unstacking		MUNSKERR	
MPU or default memory map mismatch during lazy floating-point state preservation		MLSPERR	
Bus error during exception stacking	BusFault	STKERR	*BusFault Status Register* on page 4-494
Bus error during exception unstacking		UNSTKERR	
Bus error during instruction prefetch		IBUSERR	
Bus error during lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	UsageFault	NOCP	*UsageFault Status Register* on page 4-492
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state [i]		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Stack overflow flag		STKOF	
Divide By 0		DIVBYZERO	
Lazy state error flag	SecureFault	LSERR	*Secure Fault Status Register* on page 4-560
Lazy state preservation error flag		LSPERR	
Invalid transition flag		INVTRAN	
Attribution unit violation flag		AUVIOL	
Invalid integrity signature flag		INVIS	
Invalid entry point		INVEP	
Invalid exception return flag		INVER	
Instruction cache ECC, Data cache ECC, and TCM ECC	BusFault	IS	*BusFault Status Register* on page 4-494 and *4.3.2 Auxiliary Fault Status Register* on page 4-476

h Occurs on an access to an XN region even if the processor does not include an MPU or the MPU is disabled.
i Attempting to use an instruction set other than the T32 instruction set or returns to a non load/store-multiple instruction with ICI continuation.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.5.2 Fault escalation to HardFault

All fault exceptions other than HardFault have configurable exception priority. Software can disable execution of the handlers for these faults.

Usually, the exception priority, together with the values of the exception mask registers, determines whether the processor enters the fault handler, and whether a fault handler can preempt another fault handler.

In some situations, a fault with configurable priority is treated as a HardFault. This is called *priority escalation*, and the fault is described as *escalated to HardFault*. Escalation to HardFault occurs when:

- A fault handler or an exception handler causes a fault that is of the same priority or lower priority than the current fault or exception as the one it is servicing. This escalation to HardFault occurs because a fault handler cannot preempt itself; it must have the same priority as the current execution priority level.
- A fault occurs and the handler for that fault is not enabled.

If a BusFault occurs during a stack push when entering a BusFault handler, the BusFault does not escalate to a HardFault. This means that if a corrupted stack causes a fault, the fault handler executes even though the stack push for the handler failed. The fault handler operates but the stack contents are corrupted.

In an implementation with the Security Extension, BusFaults and fixed priority exceptions can be designated as Secure or Non-secure under the control of AIRCR.BFHFMNINS. When AIRCR.BFHFMNINS is set to:

The faults and fixed priority exceptions are also designated as Secure or Non-secure under the control of AIRCR.BFHFMNINS. When AIRCR.BFHFMNINS is set to:

0 BusFaults and fixed priority exceptions are designated as Secure. The exceptions retain the prioritization of HardFault at -1 and NMI at -2.

1 BusFaults and fixed priority exceptions are designated as Non-secure. In this case, Secure HardFault is introduced at priority -3 to ensure that faults that target Secure state are recognized.

The Non-secure state cannot inhibit BusFaults and fixed priority exceptions which target Secure state. Therefore when faults and fixed priority exceptions are Secure, Non-secure FAULTMASK (FAULTMASK_NS) only inhibits programmable priority exceptions, making it equivalent to Non-secure PRIMASK (PRIMASK_NS).

Non-secure programmable priority exceptions are mapped to the regular priority range 0-254, if AIRCR.PRIS is clear. Non-secure programmable priority exceptions are mapped to the bottom half the regular priority range, 128-255, if AIRCR.PRIS is set to 1. Therefore the FAULTMASK_NS sets the execution priority to `0x0` or `0x80`, according to AIRCR.PRIS, to mask the Non-secure programmable priority exception only.

When BusFaults and fixed priority exceptions are Secure, FAULTMASK_S sets execution priority to -1 to inhibit everything up to and including HardFault.

When BusFaults and fixed priority exceptions are designated as Non-secure, FAULTMASK_NS boosts priority to -1 to inhibit everything up to Non-secure HardFault at priority -1, while FAULTMASK_S boosts priority to -3 to inhibit all faults and fixed priority exceptions including the Secure HardFault at priority -3.

Note

Only Reset can preempt the fixed priority Secure HardFault when AIRCR.BFHFNMINS is set to 1. A Secure HardFault when AIRCR.BFHFNMINS is set to 1 can preempt any exception other than Reset. A Secure HardFault when AIRCR.BFHFNMINS is set to 0 can preempt any exception other than Reset, NMI, or another HardFault.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

In an implementation with the Security Extension, only Reset can preempt the fixed priority Secure HardFault when AIRCR.BFHFNMINS is set to 1. A Secure HardFault when AIRCR.BFHFNMINS is set to 1 can preempt any exception other than Reset. A Secure HardFault when AIRCR.BFHFNMINS is set to 0 can preempt any exception other than Reset, NMI, or another HardFault.

2.5.3 Fault status registers and fault address registers

The fault status registers indicate the cause of a fault. For BusFaults and MemManage faults, the fault address register indicates the address that is accessed by the operation that caused the fault. In an implementation with the Security Extension, for SecureFaults the fault address register also indicates the address that is accessed by the operation that caused fault.

In an implementation with the Security Extension, the processor has two physical fault address registers. One shared between the MMFAR_S, SFAR, and BFAR (only if AIRCR.BFHFNMINS is set to 0), and the other shared between the MMFAR_NS and BFAR (only if AIRCR.BFHFNMINS is set to 1). These are targeted by Secure and Non-secure faults respectively.

In an implementation without the Security Extension, the processor has one physical fault address register. It is shared between the MMFAR and BFAR.

The physical fault address register can report the address of one fault at a time, and it is updated when one of the *FARVALID bits is set for their respective faults in the associated *FSR register. Any fault that targets a fault address register with one of its *FARVALID bits already set does not update the fault address. The *FARVALID bits must be cleared before another fault address can be reported.

The following table shows the fault status and fault address registers.

Note

MMFSR, MMFAR, and UFSR are banked between Security states.

Table 2-31 Fault status and fault address registers

Handler	Status register name	Address register name	Register description
HardFault	HFSR	-	*4.3.17 HardFault Status Register* on page 4-503
MemManage	MMFSR	MMFAR	*MemManage Fault Status Register* on page 4-496 *4.3.25 MemManage Fault Address Register* on page 4-513
BusFault	BFSR	BFAR	*BusFault Status Register* on page 4-494 *4.3.5 Bus Fault Address Register* on page 4-482
UsageFault	UFSR	-	*UsageFault Status Register* on page 4-492
SecureFault	SFSR	SFAR	*Secure Fault Status Register* on page 4-560 *Secure Fault Address Register* on page 4-562
RAS fault	RFSR	-	Add Link to RFSR

2.5.4 Lockup

The processor enters a lockup state if a fault occurs when it cannot be serviced or escalated. When the processor is in lockup state, it does not execute any instructions.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The processor remains in lockup state until either:

- It is reset.
- Preemption by a higher priority exception occurs. A higher priority exception can only be:
  - An NMI from a HardFault if AIRCR.BFHFNMINS is 0.
  - Secure HardFault from a HardFault or an NMI, if AIRCR.BFHFNMINS is 1.
- It is halted by a debugger.

Only reset and debug halt can exit lockup state from Secure HardFault is AIRCR.BFHFNMINS is set to 1.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.6 Power management

The Cortex-M55 processor supports modes for sleep and deep sleep that reduce power consumption.

The SCR.SLEEPDEEPs bit selects which sleep mode is used in Secure state. For more information about the behavior of the sleep modes, see <insert reference to your description of wakeup latency, and any other relevant information>.

The SCR.SLEEPDEEP bit selects which sleep mode is used. If the Security Extension is implemented, SCR.SLEEPDEEPS controls whether SLEEPDEEP is accessible from Secure state.

2.6.1 Entering sleep mode

The system can generate spurious wakeup events. Therefore, software must be able to put the processor back into sleep mode after such an event. A program might have an idle loop to put the processor back to sleep mode.

Wait for interrupt

The *wait for interrupt* instruction, `WFI`, causes immediate entry to sleep mode unless the wakeup condition is true. When the processor executes a `WFI` instruction, it stops executing instructions and enters sleep mode.

Wait for event

The *wait for event* instruction, `WFE`, causes entry to sleep mode depending on the value of a one-bit event register.

When the processor executes a `WFE` instruction, it checks the value of the event register:

0
: The processor stops executing instructions and enters sleep mode.

1
: The processor clears the register to 0 and continues executing instructions without entering sleep mode.

If the event register is 1, it indicates that the processor must not enter sleep mode on execution of a `WFE` instruction. Typically, this is because an external event signal is asserted because of an event input, an exception occurs, or a processor in the system has executed an `SEV` instruction.

Sleep-on-exit

If the SLEEPONEXIT bit of the SCR is set to 1, when the processor completes the execution of all exception handlers, it immediately enters sleep mode without restoring the Thread context from the stack. Use this mechanism in applications that only require the processor to run when an exception occurs.

2.6.2 Wakeup from sleep mode

The conditions for the processor to wake up depend on the mechanism that causes it to enter sleep mode.

Wakeup from WFI or sleep-on-exit

Normally, the processor wakes up only when it detects an exception with sufficient priority to cause exception entry. Some embedded systems might have to execute system restore tasks after the processor wakes up, and before it executes an interrupt handler. To achieve this set the PRIMASK bit to 1 and the FAULTMASK bit to 0. If an interrupt arrives that is enabled and has a higher priority than the current exception priority, the processor wakes up but does not execute the interrupt handler until the processor sets PRIMASK to zero.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Wakeup from WFE

The processor wakes up from WFE if:

- It detects an exception with sufficient priority to cause exception entry.
- It detects an external event signal.

In addition, if the SEVONPEND bit in the SCR is set to 1, any new pending interrupt triggers an event and wakes up the processor, even if the interrupt is disabled or has insufficient priority to cause exception entry.

2.6.3 The Wakeup Interrupt Controllers

When the processor core is in deepsleep mode and the *Nested Vectored Interrupt Controller* (NVIC) does not control wakeup, the *Wake-up Interrupt Controllers* (WICs) are responsible for latching pending interrupts and detecting wakeup conditions.

The Cortex-M55 processor supports two types of WIC:

- The *Internal Wake-up Interrupt Controller* (IWIC), located inside the processor.
- The *External Wake-up Interrupt Controller* (EWIC), which is an external peripheral.

The IWIC is not programmable, and does not have any registers or user interface. It operates entirely from hardware signals.

When entering IWIC sleep, the processor transfers any external interrupts already pended in the NVIC to the WIC. This prevents interrupts at a lower priority than the current thread being lost if the processor is powered down. Any interrupts pended before or during WIC Sleep are transferred back to the NVIC when a wakeup event occurs. The WIC that is used is determined by the **WICCONTROL** signal.

Both the IWIC and EWIC have their own clock, reset, and wake-up signals.

A system with the Cortex-M55 processor can choose to:

- Not implement a WIC.
- Implement the IWIC only.
- Implement the EWIC only.
- Implement both the IWIC and EWIC. However, only one WIC can be used at a time for sleep operation.

2.6.4 The external event input

The processor provides an external event input signal. Peripherals can drive this signal, either to wake the processor from WFE, or to set the internal WFE event register to 1 to indicate that the processor must not enter sleep mode on a later WFE instruction.

2.6.5 Power management programming hints

ISO/IEC C cannot directly generate the `WFI` and `WFE` instructions.

The CMSIS provides the following functions for these instructions:

```
void __WFE(void) // Wait for Event
void __WFI(void) // Wait for Interrupt
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

2.7 Arm®v8.1-M MVE overview

The *M-profile Vector Extension* (MVE) is specifically for the Armv8.1-M architecture, and it provides support for various SIMD operations.

There are two variants of MVE, the integer and floating-point variant:

- MVE-I operates on 32-bit, 16-bit, and 8-bit data types, including Q7, Q15, Q31.
- MVE-F operates on half-precision and single-precision floating-point values.

In Cortex-M55, integer and half-precision and single-precision floating-point can only be included if the floating-point functionality is included. If floating-point functionality is not included, then only the integer subset of MVE can be optionally implemented.

Vector operations are divided orthogonally into:

Lanes

A section of a vector register or operation. The data that is put into a lane is referred to as an element. Multiple lanes can be executed per beat. There are four beats per vector instruction. The permitted lane widths, and lane operations per beat, are:

- For a 64-bit lane size, a beat performs half of the lane operation.
- For a 32-bit lane size, a beat performs a one lane operation.
- For a 16-bit lane size, a beat performs a two lane operations.
- For an 8-bit lane size, a beat performs a four lane operations.

Beats

The execution of a 1/4 of an MVE vector operation. Because the vector length is 128 bits, one beat of a vector add instruction equates to computing 32 bits of result data. This is independent of lane width. For example, if a lane width is 8 bits, then a single beat of a vector add instruction would perform four 8-bit additions.
The number of beats for each tick describes how much of the architectural state is updated for each Architecture tick in the common case. In a trivial implementation, an Architecture tick might be one clock cycle:

- In a single-beat system, one beat might occur for each tick.
- In a dual-beat system, two beats might occur for each tick.
- In a quad-beat system, four beats might occur for each tick.

Cortex-M55 implements a dual-beat system.

Multiple Element writes that are generated by the same vector store instruction by the same observer can be observed in any order, with the exception that writes to the same location by different Elements are observed in order of increasing vector element number.

In Cortex-M55, the *Extension Processing Unit* (EPU) handles floating-point and vector operations.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 3
The Cortex®-M55 Instruction Set, Reference Material

This chapter is the reference material for the Cortex-M55 instruction set in a User Guide. It provides general information and describes a functional group of Cortex-M55 instructions. All the instructions supported by the Cortex-M55 processor are described.

It contains the following sections:

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.1 Cortex®-M55 instructions

The T32 instruction set is supported by the Cortex-M55 processor.

Note

In the following table:

- Angle brackets, <>, enclose alternative forms of the operand.
- Braces, {}, enclose optional operands.
- The Operands column is not exhaustive.
- `Op2` is a flexible second operand that can be either a register or a constant.
- Most instructions can use an optional condition code suffix.
- This document 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. 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 your relevant assembler documentation for these details.

For more information on the instructions and operands, see the instruction descriptions.

ph conkeyref="g.arch.name.v8m/g.product.name.long"/> Cortex®-M55 instruction set summary

Table 3-1 Instruction set summary

Mnemonic	Operands	Brief description	Flags	Page
`ADC, ADCS`	`{Rd,} Rn, Op2`	Add with Carry	N,Z,C,V	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`ADD, ADDS`	`{Rd,} Rn, Op2`	Add	N,Z,C,V	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`ADD, ADDW`	`{Rd,} Rn, #imm12`	Add	-	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`ADR`	`Rd, label`	Address to Register	-	*3.32.2 ADR* on page 3-443
`AND, ANDS`	`{Rd,} Rn, Op2`	Logical AND	N,Z,C	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`ASR, ASRS`	`Rd, Rm, <Rs|#n>`	Arithmetic Shift Right	N,Z,C	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`B {cond}`	`label`	Branch {conditionally}	-	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BFC`	`Rd, #lsb, #width`	Bit Field Clear	-	*3.18.2 BFC and BFI* on page 3-208
`BFI`	`Rd, Rn, #lsb, #width`	Bit Field Insert	-	*3.18.2 BFC and BFI* on page 3-208
`BIC, BICS`	`{Rd,} Rn, Op2`	Bit Clear	N,Z,C	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`BKPT`	`#imm8`	Breakpoint	-	*3.31.2 BKPT* on page 3-426
`BL`	`label`	Branch with Link	-	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BLX`	`Rm`	Branch indirect with Link and Exchange	-	*3.19.2 B, BL, BX, and BLX* on page 3-211

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
BLXNS	Rm	Branch indirect with Link and Exchange, Non-secure	-	*3.19.3 BXNS and BLXNS* on page 3-213
BX	Rm	Branch and Exchange	-	*3.19.2 B, BL, BX, and BLX* on page 3-211
BXNS	Rm	Branch and Exchange, Non-secure	-	*3.19.3 BXNS and BLXNS* on page 3-213
CBNZ	Rn, label	Compare and Branch on Non Zero	-	*3.19.4 CBZ and CBNZ* on page 3-214
CBZ	Rn, label	Compare and Branch on Zero	-	*3.19.4 CBZ and CBNZ* on page 3-214
CDP, CDP2	{*cond*} *coproc*, #*op1*, *Rt*, *CRn*, *CRm*{, #*op2*}	Coprocessor Data Processing	-	*3.14.3 CDP and CDP2* on page 3-164
CLREX	-	Clear Exclusive	-	*3.32.13 CLREX* on page 3-463
CLZ	Rd, Rm	Count Leading Zeros	-	*3.13.5 CLZ* on page 3-134
CMN	Rn, Op2	Compare Negative	N,Z,C,V	*3.13.6 CMP and CMN* on page 3-135
CMP	Rn, Op2	Compare	N,Z,C,V	*3.13.6 CMP and CMN* on page 3-135
CPSID	i	Change Processor State, Disable Interrupts	-	*3.31.3 CPS* on page 3-427
CPSIE	i	Change Processor State, Enable Interrupts	-	*3.31.3 CPS* on page 3-427
DMB	{opt}	Data Memory Barrier	-	*3.31.5 DMB* on page 3-428
DSB	{opt}	Data Synchronization Barrier	-	*3.31.6 DSB* on page 3-429
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
FLDMDBX ,FLDMIAX	Rn	FLDMX (Decrement Before, Increment After) loads	-	*3.20.2 FLDMDBX, FLDMIAX* on page 3-222
FSTMDBX,FSTMIAX	Rn	FSTMX (Decrement Before, Increment After) stores	-	*3.20.3 FSTMDBX, FSTMIAX* on page 3-223
ISB	{opt}	Instruction Synchronization Barrier	-	*3.31.7 ISB* on page 3-430
IT	-	If Then condition block	-	*3.19.5 IT* on page 3-215
LDA	Rd, [Rn]	Load-Acquire Word		*3.32.10 LDA and STL* on page 3-458
LDAB	Rd, [Rn]	Load-Acquire Byte		*3.32.10 LDA and STL* on page 3-458
LDAEX	Rd, [Rn]	Load-Acquire Exclusive Word	-	*3.32.12 LDAEX and STLEX* on page 3-461
LDAEXB	Rd, [Rn]	Load-Acquire Exclusive Byte	-	*3.32.12 LDAEX and STLEX* on page 3-461
LDAEXH	Rd, [Rn]	Load-Acquire Exclusive Halfword	-	*3.32.12 LDAEX and STLEX* on page 3-461

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
LDAH	Rd, [Rn]	Load-Acquire Halfword	-	*3.32.10 LDA and STL* on page 3-458
LDM	Rn{!}, reglist	Load Multiple	-	*3.32.7 LDM and STM* on page 3-453
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple Decrement Before	-	*3.32.7 LDM and STM* on page 3-453
LDMIA, LDMFD	Rn{!}, reglist	Load Multiple, Increment After	-	*3.32.7 LDM and STM* on page 3-453
LDR	Rt, [Rn, Rm {, LSL #shift}]	Load Register Word (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
LDR	Rt, label	Load Register Word (literal)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDR, LDRT	Rt, [Rn, #offset]	Load Register Word (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
LDRB	Rt, [Rn, Rm {, LSL #shift}]	Load Register Byte (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
LDRB	Rt, label	Load Register Byte (literal)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register Byte (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
LDRD	Rt, Rt2, [Rn, #offset]	Load Register Dual (immediate offset)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444
LDRD	Rt, Rt2, label	Load Register Dual (PC-relative)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	-	*3.32.11 LDREX and STREX* on page 3-459
LDREXB	Rt, [Rn]	Load Register Exclusive Byte	-	*3.32.11 LDREX and STREX* on page 3-459
LDREXH	Rt, [Rn]	Load Register Exclusive Halfword	-	*3.32.11 LDREX and STREX* on page 3-459
LDRH	Rt, [Rn, Rm {, LSL #shift}]	Load Register Halfword (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
LDRH	Rt, label	Load Register Halfword (literal)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register Halfword (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
LDRSB	Rt, [Rn, Rm {, LSL #shift}]	Load Register Signed Byte (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
LDRSB	Rt, label	Load Register Signed Byte (PC-relative)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register Signed Byte (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
LDRSH	Rt, [Rn, Rm {, LSL #shift}]	Load Register Signed Halfword (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
LDRSH	Rt, label	Load Register Signed Halfword (PC-relative)	-	*3.32.6 LDR, PC-relative* on page 3-451
LDRSH, LDRSHT	Rt, [Rn,#offset]	Load Register Signed Halfword (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
MCR,MCR2	{*cond*} *coproc*, #*opc1*, *Rt*, *CRn*, *CRm*{, #*opc2*}	Move to Coprocessor from Register	-	*3.14.4 MCR and MCR2* on page 3-165
MCRR,MCRR2	{*cond*} *coproc*, #*opc1*, *Rt*, *Rt2*, *CRm*	Move to Coprocessor from two Registers	-	*3.14.5 MCRR and MCRR2* on page 3-166
MLA	Rd, Rn, Rm, Ra	Multiply Accumulate	-	*3.15.2 MUL, MLA, and MLS* on page 3-171
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract	-	*3.15.2 MUL, MLA, and MLS* on page 3-171
MOV, MOVS	Rd, Op2	Move	N,Z,C	*3.13.7 MOV and MVN* on page 3-136
MOV, MOVS	Rd, Rm	Move (register)	N,Z	*3.13.7 MOV and MVN* on page 3-136
MOVT	Rd, #imm16	Move Top	-	*3.13.8 MOVT* on page 3-138
MOVW	Rd, #imm16	Move 16-bit constant	N,Z,C	*3.13.7 MOV and MVN* on page 3-136
MRC,MRC2	{*cond*} *coproc*, #*opc1*, *Rt*, *CRn*, *CRm*{, #*opc2*}	Move to Register from | Coprocessor	-	*3.14.6 MRC and MRC2* on page 3-167
MRRC,MRRC2	{*cond*} *coproc*, #*opc1*, *Rt*, *Rt2*, *CRm*	Move to two Registers from Coprocessor.	-	*3.14.7 MRRC and MRRC2* on page 3-168
MRS	Rd, spec_reg	Move from Special Register to general register	-	*3.31.8 MRS* on page 3-431
MSR	spec_reg, Rn	Move from general register to Special Register	-	*3.31.9 MSR* on page 3-432
MUL, MULS	{Rd,} Rn, Rm	Multiply	N,Z	*3.15.2 MUL, MLA, and MLS* on page 3-171
MVN, MVNS	Rd, Op2	Bitwise NOT	N,Z,C	*3.13.7 MOV and MVN* on page 3-136
NOP	-	No Operation	-	*3.31.10 NOP* on page 3-433
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
PKHTB, PKHBT	{Rd,} Rn, Rm, {, Op2}	Pack Halfword	-	*3.17.2 PKHBT and PKHTB* on page 3-201

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
PLD	[Rn {, #offset}]	Preload Data	-	*3.32.8 PLD* on page 3-455
POP	reglist	Pop registers from stack	-	*3.32.9 PUSH and POP* on page 3-456
PUSH	reglist	Push registers onto stack	-	*3.32.9 PUSH and POP* on page 3-456
QADD	{Rd,} Rn, Rm	Saturating Add	Q	*3.16.4 QADD and QSUB* on page 3-193
QADD16	{Rd,} Rn, Rm	Saturating Add 16	-	*3.16.4 QADD and QSUB* on page 3-193
QADD8	{Rd,} Rn, Rm	Saturating Add 8	-	*3.16.4 QADD and QSUB* on page 3-193
QASX	{Rd,} Rn, Rm	Saturating Add and Subtract with Exchange	-	*3.16.5 QASX and QSAX* on page 3-195
QDADD	{Rd,} Rn, Rm	Saturating Double and Add	Q	*3.16.6 QDADD and QDSUB* on page 3-196
QDSUB	{Rd,} Rn, Rm	Saturating Double and Subtract	Q	*3.16.6 QDADD and QDSUB* on page 3-196
QSAX	{Rd,} Rn, Rm	Saturating Subtract and Add with Exchange	-	*3.16.5 QASX and QSAX* on page 3-195
QSUB	{Rd,} Rn, Rm	Saturating Subtract	Q	*3.16.4 QADD and QSUB* on page 3-193
QSUB16	{Rd,} Rn, Rm	Saturating Subtract 16	-	*3.16.4 QADD and QSUB* on page 3-193
QSUB8	{Rd,} Rn, Rm	Saturating Subtract 8	-	*3.16.4 QADD and QSUB* on page 3-193
RBIT	Rd, Rn	Reverse Bits	-	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
REV	Rd, Rn	Reverse byte order in a word	-	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
REV16	Rd, Rn	Reverse byte order in each halfword	-	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
SADD16	{Rd,} Rn, Rm	Signed Add 16	GE	*3.13.10 SADD16 and SADD8* on page 3-140
SADD8	{Rd,} Rn, Rm	Signed Add 8	GE	*3.13.10 SADD16 and SADD8* on page 3-140
SASX	{Rd,} Rn, Rm	Signed Add and Subtract with Exchange	GE	*3.13.11 SASX and SSAX* on page 3-142
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	-	*3.18.3 SBFX and UBFX* on page 3-209

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
SDIV	{Rd,} Rn, Rm	Signed Divide	-	*3.15.3 SDIV and UDIV* on page 3-172
SEL	{Rd,} Rn, Rm	Select bytes	GE	*3.13.12 SEL* on page 3-144
SEV	-	Send Event	-	*3.31.11 SEV* on page 3-434
SG	-	Secure Gateway	-	*3.31.12 SG* on page 3-435
SHADD16	{Rd,} Rn, Rm	Signed Halving Add 16	-	*3.13.13 SHADD16 and SHADD8* on page 3-145
SHADD8	{Rd,} Rn, Rm	Signed Halving Add 8	-	*3.13.13 SHADD16 and SHADD8* on page 3-145
SHASX	{Rd,} Rn, Rm	Signed Halving Add and Subtract with Exchange	-	*3.13.14 SHASX and SHSAX* on page 3-146
SHSAX	{Rd,} Rn, Rm	Signed Halving Subtract and Add with Exchange	-	*3.13.14 SHASX and SHSAX* on page 3-146
SHSUB16	{Rd,} Rn, Rm	Signed Halving Subtract 16	-	*3.13.15 SHSUB16 and SHSUB8* on page 3-147
SHSUB8	{Rd,} Rn, Rm	Signed Halving Subtract 8	-	*3.13.15 SHSUB16 and SHSUB8* on page 3-147
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate halfwords	Q	*3.15.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-173
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Dual	Q	*3.15.5 SMLAD and SMLADX* on page 3-175
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long (32 × 32 + 64), 64-bit result	-	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long, halfwords	-	*3.15.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-177
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long Dual	-	*3.15.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-177
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate, word by halfword	Q	*3.15.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-173
SMLSD, SMLSDX	Rd, Rn, Rm, Ra	Signed Multiply Subtract Dual	Q	*3.15.7 SMLSD and SMLSLD* on page 3-179
SMLSLD, SMLSLDX	RdLo, RdHi, Rn, Rm	Signed Multiply Subtract Long Dual	-	*3.15.7 SMLSD and SMLSLD* on page 3-179
SMMLA, SMMLAR	Rd, Rn, Rm, Ra	Signed Most Significant Word Multiply Accumulate	-	*3.15.8 SMMLA and SMMLS* on page 3-181
SMMLS, SMMLSR	Rd, Rn, Rm, Ra	Signed Most Significant Word Multiply Subtract	-	*3.15.8 SMMLA and SMMLS* on page 3-181

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
SMMUL, SMMULR	Rd, Rn, Rm	Signed Most Significant Word Multiply	-	*3.15.9 SMMUL* on page 3-183
SMUAD, SMUADX	{Rd,} Rn, Rm	Signed Dual Multiply Add	Q.	*3.15.10 SMUAD and SMUSD* on page 3-184
SMULBB, SMULBT, SMULTB, SMULTT	{Rd,} Rn, Rm	Signed Multiply (halfwords)	-	*3.15.11 SMUL and SMULW* on page 3-186
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply Long (32 × 32), 64-bit result	-	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
SMULWB, SMULWT	{Rd,} Rn, Rm	Signed Multiply word by halfword	-	*3.15.11 SMUL and SMULW* on page 3-186
SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed Dual Multiply Subtract	-	*3.15.10 SMUAD and SMUSD* on page 3-184
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q	*3.16.2 SSAT and USAT* on page 3-191
SSAT16	Rd, #n, Rm	Signed Saturate 16	Q	*3.16.3 SSAT16 and USAT16* on page 3-192
SSAX	{Rd,} Rn, Rm	Signed Subtract and Add with Exchange	GE	*3.13.11 SASX and SSAX* on page 3-142
SSUB16	{Rd,} Rn, Rm	Signed Subtract 16	GE	*3.13.16 SSUB16 and SSUB8* on page 3-148
SSUB8	{Rd,} Rn, Rm	Signed Subtract 8	GE	*3.13.16 SSUB16 and SSUB8* on page 3-148
STL	Rt, [Rn]	Store-Release Word	-	*3.32.10 LDA and STL* on page 3-458
STLB	Rt, [Rn]	Store-Release Byte	-	*3.32.10 LDA and STL* on page 3-458
STLEX	Rt, Rt [Rn]	Store-Release Exclusive Word	-	*3.32.12 LDAEX and STLEX* on page 3-461
STLEXB	Rt, Rt [Rn]	Store-Release Exclusive Byte	-	*3.32.12 LDAEX and STLEX* on page 3-461
STLEXH	Rt, Rt [Rn]	Store-Release Exclusive Halfword	-	*3.32.12 LDAEX and STLEX* on page 3-461
STLH	Rt, [Rn]	Store-Release Halfword	-	*3.32.10 LDA and STL* on page 3-458
STM	Rn{!}, reglist	Store Multiple	-	*3.32.7 LDM and STM* on page 3-453
STMDB, STMEA	Rn{!}, reglist	Store Multiple Decrement Before	-	*3.32.7 LDM and STM* on page 3-453
STMIA, STMFD	Rn{!}, reglist	Store Multiple Increment After	-	*3.32.7 LDM and STM* on page 3-453
STR	Rt, [Rn, Rm {, LSL #shift}]	Store Register Word (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
STR, STRT	Rt, [Rn, #offset]	Store Register Word (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
STRB	Rt, [Rn, Rm {, LSL #shift}]	Store Register Byte (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
`STRB, STRBT`	`Rt, [Rn, #offset]`	Store Register Byte (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
`STRD`	`Rt, Rt2, [Rn, #offset]`	Store Register Dual two words	-	*3.32.3 LDR and STR, immediate offset* on page 3-444
`STREX`	`Rd, Rt, [Rn, #offset]`	Store Register Exclusive	-	*3.32.11 LDREX and STREX* on page 3-459
`STREXB`	`Rd, Rt, [Rn]`	Store Register Exclusive Byte	-	*3.32.11 LDREX and STREX* on page 3-459
`STREXH`	`Rd, Rt, [Rn]`	Store Register Exclusive Halfword	-	*3.32.11 LDREX and STREX* on page 3-459
`STRH`	`Rt, [Rn, Rm {, LSL #shift}]`	Store Register Halfword (register offset)	-	*3.32.4 LDR and STR, register offset* on page 3-447
`STRH, STRHT`	`Rt, [Rn, #offset]`	Store Register Halfword (immediate offset, unprivileged)	-	*3.32.3 LDR and STR, immediate offset* on page 3-444, *3.32.5 LDR and STR, unprivileged* on page 3-449
`SUB, SUBS`	`{Rd,} Rn, Op2`	Subtract	N,Z,C,V	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`SUB, SUBW`	`{Rd,} Rn, #imm12`	Subtract	-	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`SVC`	`#imm`	Supervisor Call	-	*3.31.13 SVC* on page 3-436
`SXTAB`	`{Rd,} Rn, Rm {,ROR #n}`	Sign extend 8 bits to 32 and Add	-	*3.17.3 SXTA and UXTA* on page 3-203
`SXTAB16`	`{Rd,} Rn, Rm {,ROR #n}`	Sign extend two 8-bit values to 16 and Add	-	*3.17.3 SXTA and UXTA* on page 3-203
`SXTAH`	`{Rd,} Rn, Rm {,ROR #n}`	Sign extend 16 bits to 32 and Add	-	*3.17.3 SXTA and UXTA* on page 3-203
`SXTB`	`Rd, Rm {,ROR #n}`	Sign extend 8 bits to 32	-	*3.17.4 SXT and UXT* on page 3-205
`SXTB16`	`{Rd,} Rm {,ROR #n}`	Sign extend 8 bits to 16	-	*3.17.4 SXT and UXT* on page 3-205
`SXTH`	`{Rd,} Rm {,ROR #n}`	Sign extend a Halfword to 32	-	*3.17.4 SXT and UXT* on page 3-205
`TBB`	`[Rn, Rm]`	Table Branch Byte	-	*3.19.6 TBB and TBH* on page 3-217
`TBH`	`[Rn, Rm, LSL #1]`	Table Branch Halfword	-	*3.19.6 TBB and TBH* on page 3-217
`TEQ`	`Rn, Op2`	Test Equivalence	N,Z,C	*3.13.17 TST and TEQ* on page 3-150
`TST`	`Rn, Op2`	Test	N,Z,C	*3.13.17 TST and TEQ* on page 3-150
`TT`	`Rd, [Rn]`	Test Target	-	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
`TTA`	`Rd, [Rn]`	Test Target Alternate Domain	-	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
TTAT	Rd, [Rn]	Test Target Alternate Domain Unprivileged	-	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
TTT	Rd, [Rn]	Test Target Unprivileged	-	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
UADD16	{Rd,} Rn, Rm	Unsigned Add 16	GE	*3.13.18 UADD16 and UADD8* on page 3-151
UADD8	{Rd,} Rn, Rm	Unsigned Add 8	GE	*3.13.18 UADD16 and UADD8* on page 3-151
UASX	{Rd,} Rn, Rm	Unsigned Add and Subtract with Exchange	GE	*3.13.19 UASX and USAX* on page 3-153
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	-	*3.18.3 SBFX and UBFX* on page 3-209
UDF	{c}{q} {#}imm	Permanently Undefined.	-	*3.31.15 UDF* on page 3-439
UDIV	{Rd,} Rn, Rm	Unsigned Divide	-	*3.15.3 SDIV and UDIV* on page 3-172
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16	-	*3.13.20 UHADD16 and UHADD8* on page 3-155
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8	-	*3.13.20 UHADD16 and UHADD8* on page 3-155
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange	-	*3.13.21 UHASX and UHSAX* on page 3-156
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange	-	*3.13.21 UHASX and UHSAX* on page 3-156
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16	-	*3.13.22 UHSUB16 and UHSUB8* on page 3-158
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8	-	*3.13.22 UHSUB16 and UHSUB8* on page 3-158
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long (32 × 32 + 32 + 32), 64-bit result	-	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Long (32 × 32 + 64), 64-bit result	-	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Long (32 × 32), 64-bit result	-	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	-	*3.16.8 UQADD and UQSUB* on page 3-198
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	-	*3.16.8 UQADD and UQSUB* on page 3-198
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	-	*3.16.7 UQASX and UQSAX* on page 3-197
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	-	*3.16.7 UQASX and UQSAX* on page 3-197

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	-	*3.16.8 UQADD and UQSUB* on page 3-198
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	-	*3.16.8 UQADD and UQSUB* on page 3-198
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	-	*3.13.23 USAD8* on page 3-159
USADA8	Rd, Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	-	*3.13.24 USADA8* on page 3-160
USAT	Rd, #n, Rm{,shift #s}, Ra	Unsigned Saturate	Q	*3.16.2 SSAT and USAT* on page 3-191
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q	*3.16.3 SSAT16 and USAT16* on page 3-192
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE	*3.13.19 UASX and USAX* on page 3-153
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE	*3.13.25 USUB16 and USUB8* on page 3-161
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE	*3.13.25 USUB16 and USUB8* on page 3-161
UXTAB	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend 8 bits to 32 and Add	-	*3.17.3 SXTA and UXTA* on page 3-203
UXTAB16	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend two 8-bit values to 16 and Add	-	*3.17.3 SXTA and UXTA* on page 3-203
UXTAH	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend and Add Halfword	-	*3.17.3 SXTA and UXTA* on page 3-203
UXTB	Rd, Rm {,ROR #n}	Unsigned zero-extend Byte	-	*3.17.4 SXT and UXT* on page 3-205
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned zero-extend Byte 16	-	*3.17.4 SXT and UXT* on page 3-205
UXTH	Rd, Rm {,ROR #n}	Unsigned zero-extend Halfword	-	*3.17.4 SXT and UXT* on page 3-205
VABS	.F32 Sd, Sm	Floating-point Absolute	-	*3.20.4 VABS* on page 3-224
VADD	.F32 {Sd,} Sn, Sm	Floating-point Add	-	*3.20.5 VADD* on page 3-225
VCMP	.F32 Sd, <Sm|#0.0>	Compare two floating-point registers, or one floating-point register and zero	N,Z,C,V	*3.20.6 VCMP and VCMPE* on page 3-226
VCMPE	.F32 Sd, <Sm|#0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	N,Z,C,V	*3.20.6 VCMP and VCMPE* on page 3-226
VCVT	.F32.Tm <Sd>, Sm	Convert from floating-point to integer	-	*3.20.7 VCVT and VCVTR between floating-point and integer* on page 3-227
VCVT	.Td.F32 Sd, Sd, #fbits	Convert from floating-point to fixed point	-	*3.20.8 VCVT between floating-point and fixed-point* on page 3-228

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
VCVTA	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding to nearest with Ties Away	-	*3.20.36 VCVTA, VCVTM VCVTN, and VCVTP* on page 3-256
VCVTB VCVTT	.F32.F16 Sd, Sm	Convert half-precision value to single-precision or double-precision	-	*3.20.37 VCVTB and VCVTT* on page 3-257
VCVTB VCVTT	.F16.F32 Sd, Sm	Convert single-precision or double-precision register to half-precision	-	*3.20.37 VCVTB and VCVTT* on page 3-257
VCVTM	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding towards Minus infinity	-	*3.20.36 VCVTA, VCVTM VCVTN, and VCVTP* on page 3-256
VCVTN	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding to nearest with Ties to even	-	*3.20.36 VCVTA, VCVTM VCVTN, and VCVTP* on page 3-256
VCVTP	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding towards Plus infinity	-	*3.20.36 VCVTA, VCVTM VCVTN, and VCVTP* on page 3-256
VCVTR	.Tm.F32 <Sd>, Sm	Convert between floating-point and integer with rounding.	-	*3.20.7 VCVT and VCVTR between floating-point and integer* on page 3-227
VDIV	.F32 {Sd,} Sn, Sm	Floating-point Divide	-	*3.20.9 VDIV* on page 3-229
VFMA	.F32 {Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate	-	*3.20.10 VFMA and VFMS* on page 3-230
VFMS	.F32 {Sd,} Sn, Sm	Floating-point Fused Multiply Subtract	-	*3.20.10 VFMA and VFMS* on page 3-230
VFNMA	.F32 {Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate	-	*3.20.11 VFNMA and VFNMS* on page 3-231
VFNMS	.F32 {Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract	-	*3.20.11 VFNMA and VFNMS* on page 3-231
VLDM	{mode}{.size} Rn{!}, list	Floating-point Load Multiple extension registers	-	*3.20.12 VLDM* on page 3-232
VLDR	.F32 Sd, [<Rn> {, #offset}]	Floating-point Load an extension register from memory (immediate)	-	*3.20.13 VLDR* on page 3-233
VLDR	.F32 Sd, <label>	Load an extension register from memory	-	*3.20.13 VLDR* on page 3-233
VLDR	.F32 Sd, [PC,#-0]	Load an extension register from memory	-	*3.20.13 VLDR* on page 3-233

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
VLLDM	<c> Rn	Floating-point Lazy Load multiple	-	*3.20.14 VLLDM* on page 3-234
VLSTM	<c> Rn	Floating-point Lazy Store multiple	-	*3.20.15 VLSTM* on page 3-235
VMAXNM	.F32 Sd, Sn, Sm	Maximum of two floating-point numbers with IEEE754-2008 NaN handling	-	*3.20.38 VMAXNM and VMINNM* on page 3-258
VMINNM	.F32 Sd, Sn, Sm	Minimum of two floating-point numbers with IEEE754-2008 NaN handling	-	*3.20.38 VMAXNM and VMINNM* on page 3-258
VMLA	.F32 Sd, Sn, Sm	Floating-point Multiply Accumulate	-	*3.20.16 VMLA and VMLS* on page 3-236
VMLS	.F32 Sd, Sn, Sm	Floating-point Multiply Subtract	-	*3.20.16 VMLA and VMLS* on page 3-236
VMOV	<Sn|Rt>, <Rt|Sn>	Copy core register to single-precision	-	*3.20.20 VMOV core register to single-precision* on page 3-240
VMOV	<Sm|Rt>, <Sm1|Rt2>, <Rt|Sm>, <Rt2|Sm1>	Copy two core registers to two single-precision	-	*3.20.21 VMOV two core registers to two single-precision registers* on page 3-241
VMOV	{.size} Dd[x], Rt	Copy core register to scalar	-	*3.20.23 VMOV core register to scalar* on page 3-243
VMOV	{.dt} Rt, Dn[x]	Copy scalar to core register	-	*3.20.19 VMOV scalar to core register* on page 3-239
VMOV	.F32 Sd, #immm	Floating-point Move immediate	-	*3.20.17 VMOV Immediate* on page 3-237
VMOV	.F32 Sd, Sd, Sm	Copies the contents of one register to another	-	*3.20.18 VMOV Register* on page 3-238
VMOV	<Dm|Rt>, <Rt|Rt2>, <Rt2|Dm>	Floating-point Move transfers two words between two core registers and a doubleword register	-	*3.20.22 VMOV two core registers and a double-precision register* on page 3-242
VMRS	Rt, FPSCR	Move to core register from floating-point Special Register	N,Z,C,V	*3.20.24 VMRS* on page 3-244
VMSR	FPSCR, Rt	Move to floating-point Special Register from core register	-	*3.20.25 VMSR* on page 3-245
VMUL	.F32 {Sd,} Sn, Sm	Floating-point Multiply	-	*3.20.26 VMUL* on page 3-246
VNEG	.F32 Sd, Sm	Floating-point Negate	-	*3.20.27 VNEG* on page 3-247
VNMLA	.F32 Sd, Sn, Sm	Floating-point Multiply Accumulate and Negate	-	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
VNMLS	.F32 Sd, Sn, Sm	Floating-point Multiply, Subtract and Negate	-	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248
VNMUL	.F32 {Sd,} Sn, Sm	Floating-point Multiply and Negate	-	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248
VPOP	{.size} list	Load multiple consecutive floating-point registers from the stack	-	*3.20.29 VPOP* on page 3-249
VPUSH	{.size} list	Store multiple consecutive floating-point registers to the stack	-	*3.20.30 VPUSH* on page 3-250
VRINTA	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties Away	-	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VRINTM	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Minus infinity	-	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VRINTN	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties to even	-	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VRINTP	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Plus infinity	-	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VRINTR	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding towards value specified in FPSCR	-	*3.20.39 VRINTR and VRINTX* on page 3-259
VRINTX	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding specified in FPSCR	-	*3.20.39 VRINTR and VRINTX* on page 3-259
VRINTZ	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding towards Zero	-	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VSEL	.F32 Sd, Sn, Sm	Select register, alternative to a pair of conditional VMOV	-	*3.20.35 VSEL* on page 3-255
VSQRT	.F32 Sd, Sm	Calculates floating-point Square Root	-	*3.20.31 VSQRT* on page 3-251
VSTM	{mode}{.size} Rn{!}, list	Floating-point Store Multiple	-	*3.20.32 VSTM* on page 3-252

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-1 Instruction set summary (continued)

Mnemonic	Operands	Brief description	Flags	Page
VSTR	.F32 Sd, [Rn{, #offset}]	Floating-point Store Register stores an extension register to memory	-	*3.20.33 VSTR* on page 3-253
VSUB	F32 {Sd,} Sn, Sm	Floating-point Subtract	-	*3.20.34 VSUB* on page 3-254
WFE	-	Wait For Event	-	*3.31.16 WFE* on page 3-440
WFI	-	Wait For Interrupt	-	*3.31.17 WFI* on page 3-441
YIELD	-	Suspend task	-	*3.31.18 YIELD* on page 3-441

Arm®v8.1-M Cortex®-M55 instruction set summary

- *3.22.1 List of Arm®v8.1-M branch and loop instructions* on page 3-295
- *3.23.1 List of Arm®v8.1-M comparison and vector predication operations instructions* on page 3-301
- *3.27.1 List of Arm®v8.1-M vector floating-point conversion and rounding operation instructions* on page 3-343
- *3.24.1 List of Arm®v8.1-M vector load and store operations instructions* on page 3-325
- *3.28.1 List of Arm®v8.1-M security instructions* on page 3-348
- *3.25.1 List of Arm®v8.1-M vector move operation instructions* on page 3-339
- *3.29.1 List of Arm®v8.1-M vector arithmetic instructions* on page 3-350
- *3.30.1 List of Arm®v8.1-M vector bitwise operations instructions* on page 3-409

3.1.1 Binary compatibility with other Cortex processors

The processor implements the T32 instruction set and features provided by the Armv8-M architecture profile. There are restrictions on moving code designed for processors that are implementations of the Armv6-M or Armv7-M architectures.

If code designed for other Cortex-M processors relies on memory protection, it cannot be moved to the Cortex-M55 processor. In this case, the memory protection scheme and driver code must be updated from PMSAv7 to PMSAv8.

If Cortex-M55 is configured without floating-point, any Armv7-M code that uses floating-point arithmetic must be recompiled to use a software library, or DP emulation if supported by the tools.

To ensure a smooth transition, Arm recommends that code designed to operate on other Cortex-M profile processor architectures obey the following rules and that you configure the *Configuration and Control Register* (CCR) appropriately:

- Use word transfers only to access registers in the NVIC and *System Control Space* (SCS).
- Treat all unused SCS registers and register fields on the processor as Do-Not-Modify.
- Configure the following fields in the CCR:
  - — STKALIGN bit to 1.
  - — UNALIGN_TRP bit to 1.
  - — Leave all other bits in the CCR register at their original value.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.2 CMSIS functions

ISO/IEC C code cannot directly access some Cortex-M55 processor instructions. Instead, intrinsic functions that are provided by the CMSIS or a C compiler are used to generate them. If a C compiler does not support an appropriate intrinsic function, you might have to use inline assembler to access some instructions.

3.2.1 List of CMSIS functions to generate some processor instructions

List of intrinsic functions that are provided to generate instructions that ISO/IEC C code cannot directly access.

Table 3-2 CMSIS functions to generate some Cortex-M55 processor instructions

Instruction	CMSIS function
`BKPT`	`void __BKPT`
`CLREX`	`void __CLREX`
`CLZ`	`uint8_t __CLZ (uint32_t value)`
`CPSID F`	`void __disable_fault_irq(void)`
`CPSID I`	`void __disable_irq(void)`
`CPSIE F`	`void __enable_fault_irq(void)`
`CPSIE I`	`void __enable_irq(void)`
`DMB`	`void __DMB(void)`
`DSB`	`void __DSB(void)`
`ISB`	`void __ISB(void)`
`LDA`	`uint32_t __LDA (volatile uint32_t * ptr)`
`LDAB`	`uint8_t __LDAB (volatile uint8_t * ptr)`
`LDAEX`	`uint32_t __LDAEX (volatile uint32_t * ptr)`
`LDAEXB`	`uint8_t __LDAEXB (volatile uint32_t * ptr)`
`LDAEXH`	`uint16_t __LDAEXH (volatile uint32_t * ptr)`
`LDAH`	`uint32_t __LDAH (volatile uint32_t * addr)`
`LDRT`	`uint32_t __LDRT (uint32_t ptr)`
`NOP`	`void __NOP (void)`
`RBIT`	`uint32_t __RBIT(uint32_t value)`
`REV`	`uint32_t __REV(uint32_t value)`
`REV16`	`uint32_t __REV16(uint32_t value)`
`REVSH`	`int16_t __REVSH(int16_t value)`
`ROR`	`uint32_t __ROR (uint32_t value, uint32_t shift)`
`RRX`	`uint32_t __RRX (uint32_t value)`
`SEV`	`void __SEV (void)`
`STL`	`void __STL (uint32_t value, volatile uint32_t * ptr)`
`STLEX`	`uint32_t __STLEX (uint16_t value, volatile uint32_t * ptr)`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-2 CMSIS functions to generate some Cortex-M55 processor instructions (continued)

Instruction	CMSIS function
STLEXB	uint32_t __STLEXB (uint16_t value, volatile uint8_t * ptr)
STLEXH	uint32_t __STLEXH (uint16_t value, volatile uint16_t * ptr)
STLH	void __STLH (uint16_t value, volatile uint16_t * ptr)
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXB	uint32_t __STREXB (uint8_t value, uint8_t *addr)
STREXH	uint32_t __STREXH (uint16_t value, uint16_t *addr)
WFE	void __WFE(void)
WFI	void __WFI(void)

3.2.2 CMSE

CMSE is the compiler support for the Security Extension (architecture intrinsics and options) and is part of the Arm C Language (ACLE) specification.

CMSE features are required when developing software running in Secure state. This provides mechanisms to define Secure entry points and enable the tool chain to generate correct instructions or support functions in the program image.

The CMSE features are accessed using various attributes and intrinsics. Additional macros are also defined as part of the CMSE.

3.2.3 CMSIS functions to access the special registers

List of functions that are provided by the CMSIS for accessing the special registers using `MRS` and `MSR` instructions.

Table 3-3 CMSIS functions to access the special registers

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)
APSR	Read	uint32_t __get_APSR (void)
IPSR	Read	uint32_t __get_IPSR (void)

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-3 CMSIS functions to access the special registers (continued)

Special register	Access	CMSIS function
xPSR	Read	`uint32_t __get_xPSR (void)`
BASEPRI_MAX	Write	`void __set_BASEPRI_MAX (uint32_t basePri)`
FPSCR	Read	`uint32_t __get_FPSCR (void)`
	Write	`void __set_FPSCR (uint32_t fpscr)`
MSPLIM	Read	`uint32_t __get_MSPLIM (void)`
	Write	`void __set_MSPLIM (uint32_t MainStackPtrLimit)`
PSPLIM	Read	`uint32_t __get_PSPLIM (void)`
	Write	`void __set_PSPLIM (uint32_t ProcStackPtrLimit)`

3.2.4 CMSIS functions to access the Non-secure special registers

The CMSIS also provides several functions for accessing the Non-secure special registers in Secure state using `MRS` and `MSR` instructions:

Table 3-4 CMSIS intrinsic functions to access the Non-secure special registers

Special register	Access	CMSIS function
PRIMASK_NS	Read	`uint32_t __TZ_get_PRIMASK_NS (void)`
	Write	`void __TZ_set_PRIMASK_NS (uint32_t value)`
FAULTMASK_NS	Read	`uint32_t __TZ_get_FAULTMASK_NS (void)`
	Write	`void __TZ_set_FAULTMASK_NS (uint32_t value)`
CONTROL_NS	Read	`uint32_t __TZ_get_CONTROL_NS (void)`
	Write	`void __TZ_set_CONTROL_NS (uint32_t value)`
MSP_NS	Read	`uint32_t __TZ_get_MSP_NS (void)`
	Write	`void __TZ_set_MSP_NS (uint32_t TopOfMainStack)`
PSP_NS	Read	`uint32_t __TZ_get_PSP_NS (void)`
	Write	`void __TZ_set_PSP_NS (uint32_t TopOfProcStack)`
MSPLIM_NS	Read	`uint32_t __TZ_get_MSPLIM_NS (void)`
	Write	`void __TZ_set_MSPLIM_NS (uint32_t MainStackPtrLimit)`
PSPLIM_NS	Read	`uint32_t __TZ_get_PSPLIM_NS (void)`
	Write	`void __TZ_set_PSPLIM_NS (uint32_t ProcStackPtrLimit)`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.3 Operands

An instruction operand can be an Arm register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. When there is a destination register in the instruction, it is usually specified before the operands.

Operands in some instructions are flexible in that they can either be a register or a constant.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.4 Restrictions when using PC or SP

Many instructions have restrictions on whether you can use the *Program Counter* (PC) or *Stack Pointer* (SP) for the operands or destination register. See instruction descriptions for more information.

———— **Note** ————

- In an implementation with Armv8-M Security Extension, for correct operation of B{L}XNS, Rm[0] must be 0 for correct Secure to Non-secure transition.
- Bit[0] of any address you write to the PC with a `BX`, `BLX`, `LDM`, `LDR`, or `POP` instruction must be 1 for correct execution, because this bit indicates the required instruction set, and the Cortex-M55 processor only supports T32 instructions.

————————————

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.5 Flexible second operand

Many general data processing instructions have a flexible second operand. This is shown as `Operand2` in the descriptions of the syntax of each instruction.

`Operand2` can be:

- A constant.
- A register with optional shift.

This section contains the following subsections:

- *3.5.1 Constant* on page 3-106.
- *3.5.2 Register with optional shift* on page 3-106.

3.5.1 Constant

Instruction form when specifying an Operand2 constant.

```
#constant
```

where *`constant`* can be:

- Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- Any constant of the form `0x00XY00XY`.
- Any constant of the form `0xXY00XY00`.
- Any constant of the form `0xXYXYXYXY`.

Note

In these constants, `X` and `Y` are hexadecimal digits.

In addition, in a small number of instructions, *`constant`* can take a wider range of values. These are described in the individual instruction descriptions.

When an Operand2 constant is used with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to bit[31] of the constant, if the constant is greater than 255 and can be produced by shifting an 8-bit value. These instructions do not affect the carry flag if Operand2 is any other constant.

Instruction substitution

Your assembler might be able to produce an equivalent instruction in cases where you specify a constant that is not permitted.

For example, an assembler might assemble the instruction `CMP Rd, #0xFFFFFFFE` as the equivalent instruction `CMN Rd, #0x2`.

3.5.2 Register with optional shift

Instruction form when specifying an Operand2 register.

```
Rm {, shift}
```

Where:

`Rm`
: Is the register holding the data for the second operand.

`shift`
: Is an optional shift to be applied to *`Rm`*. It can be one of:

`ASR #n`

Arithmetic shift right *n* bits, $1 \leq n \leq 32$.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`LSL #n`
: Logical shift left *n* bits, $1 \leq n \leq 31$.

`LSR #n`
: Logical shift right *n* bits, $1 \leq n \leq 32$.

`ROR #n`
: Rotate right *n* bits, $1 \leq n \leq 31$.

RRX
: Shift right one bit and insert the carry flag into the most significant bit of the result.

-
: If omitted, no shift occurs, equivalent to `LSL #0`.

If you omit the shift, or specify `LSL #0`, the instruction uses the value in *Rm*.

If you specify a shift, the shift is applied to the value in *Rm*, and the resulting 32-bit value is used by the instruction. However, the contents in the register *Rm* remain unchanged. Specifying a register with shift also updates the carry flag when used with certain instructions.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.6 Right shift operations

Register right shift operations move the bits in a register right by a specified number of bits, the *shift length*.

Register shift can be performed:

- Directly by the shift instructions, and the result is written to a destination register.
- During the calculation of `Operand2` by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description for more information. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, `Rm` is the register containing the value to be shifted, and `n` is the shift length.

3.6.1 ASR

Arithmetic Shift Right 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 following figure shows an ASR #4 instruction.

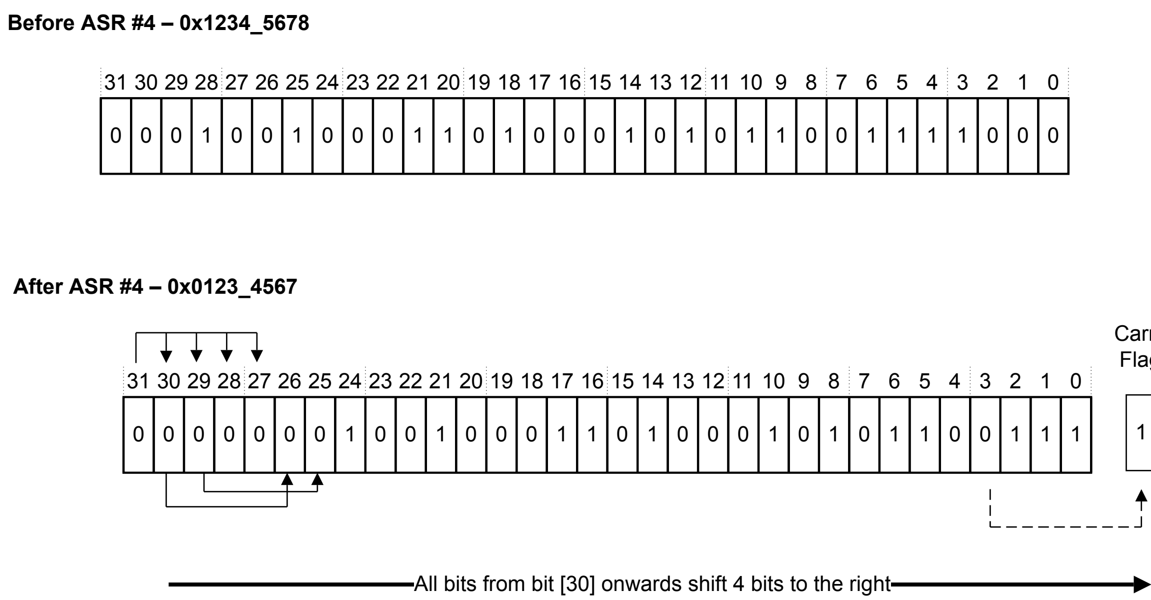


Figure 3-1 ASR #4

For more information on ASR, see MOV register encoding and LSR_C function in the *Arm®v8-M Architecture Reference Manual*.

3.6.2 ASRL

Arithmetic Shift Right Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   amount =SInt(R[m][7:0]);
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    op1    =SInt(R[dah]:R[dal]);
    result =(op1 >> amount) [63:0];
    R[dah] =result[63:32];
    R[dah] =result[31:0];
```

Example operation

If the shift `amount` is 4 and `R[dah]` and `R[dal]` are `0x12345678` and `0x87654321` respectively, then:

```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount =4;
    op1    =(0x12345678:0x87654321);
    result =((0x12345678:0x87654321) >> 4)) [63:0]; //Zeroes are added into bits [63:60]
    R[dah] =0x012345678;
    R[dal] =0x88765432;
```

The following figure shows an ASRL #4 instruction where the bits shifted out of RdaHi is shifted into RdaLo from bit [31] onwards, and the carry flag is updated to the last bit shifted out of RdaLo.

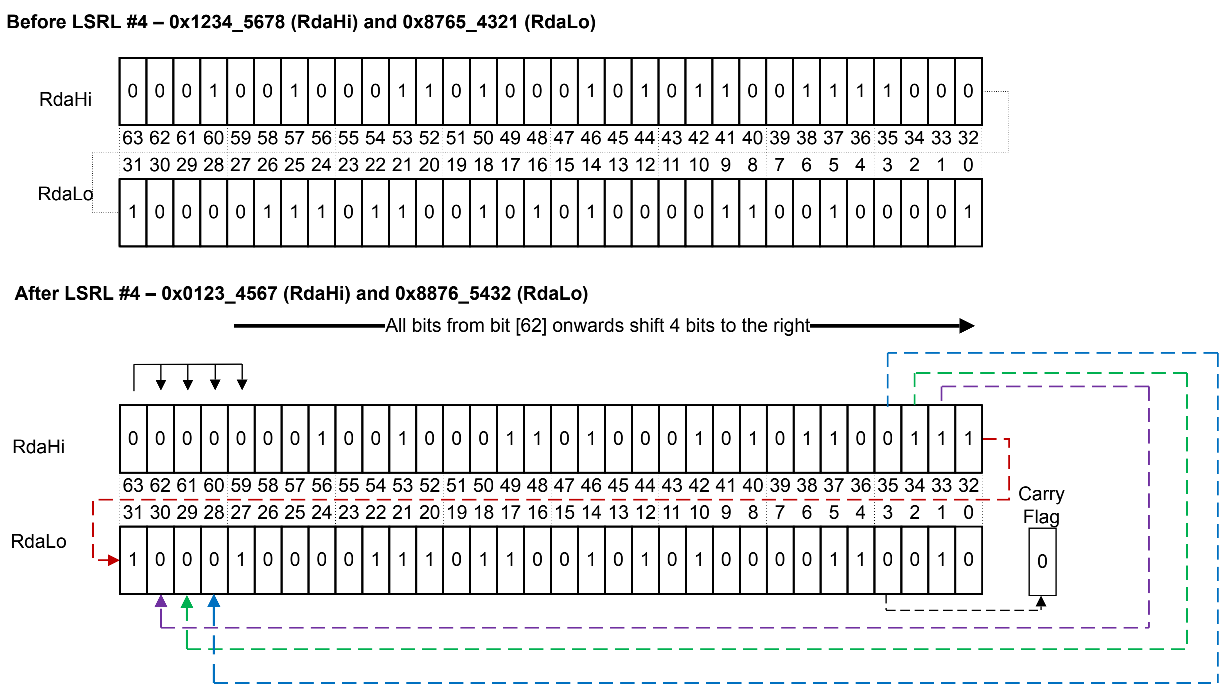


Figure 3-2 ASRL #4

For more information on ASRL, see the *Arm®v8-M Architecture Reference Manual*.

3.6.3 LSR

Logical Shift Right shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

The following figure shows an LSR #4 instruction.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

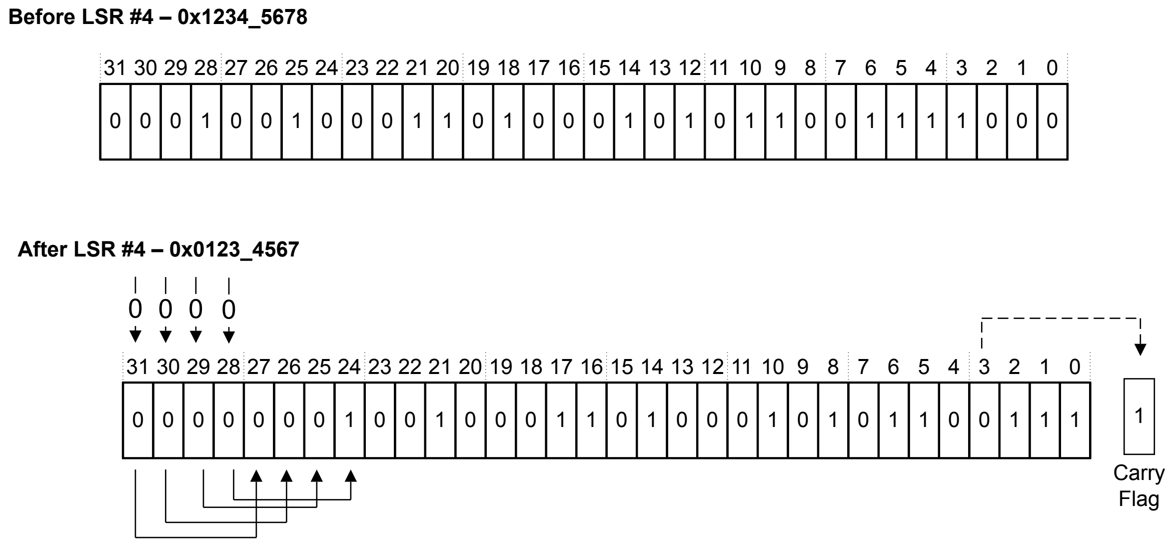


Figure 3-3 LSR #4

For more information on LSR, see MOV register encoding and LSR_C function in the *Arm®v8-M Architecture Reference Manual*.

3.6.4 LSRL

Logical Shift Right Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =UInt(R[dah]:R[dal]);
   op1    =(op1 >> amount) [63:0];
   R[dah] =result[63:32];
   R[dal] =result[31:0];
```

Example operation

If the shift `amount` is 4 and `R[dah]` and `R[dal]` are `0x12345678` and `0x87654321` respectively, then:

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =(0x12345678:0x87654321);
   op1    =((0x12345678:0x87654321) >> 4)) [63:0]; //Zeroes are added into bits [63:60]
   R[dah] =0x012345678;
   R[dal] =0x88765432;
```

The following figure shows an LSRL #4 instruction, indicating that zeros are added to the first four MSBs of RdaHi and the carry flag is updated to the last bit shifted out of RdaLo.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

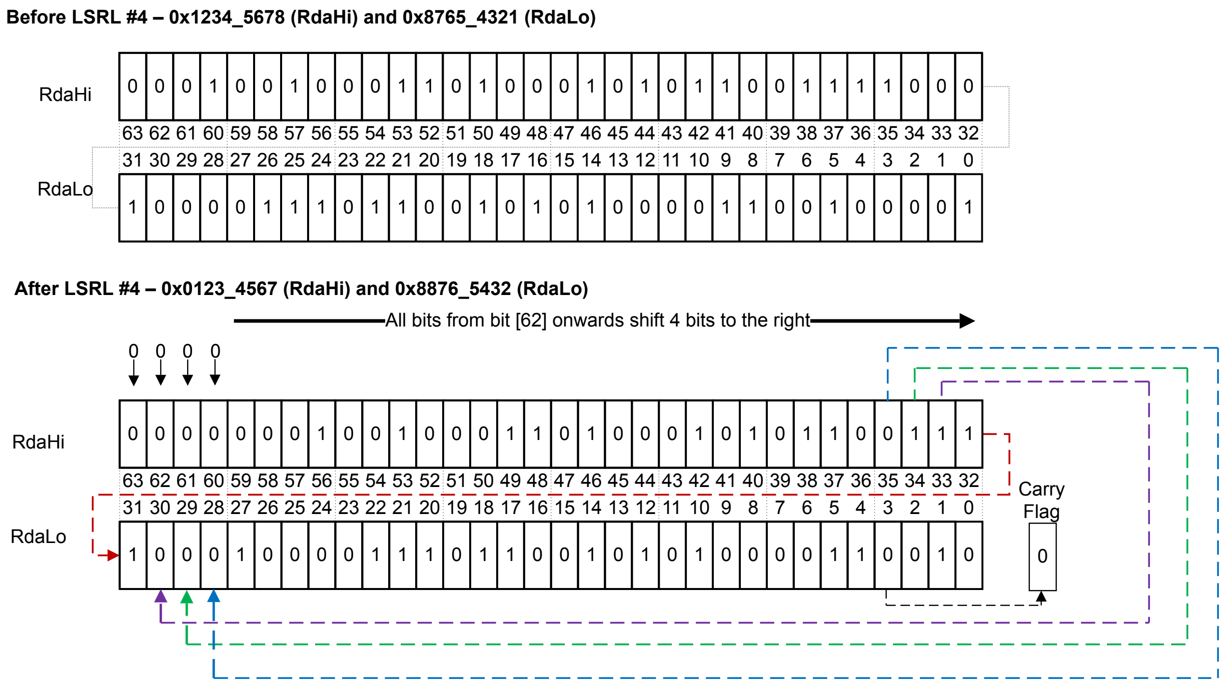


Figure 3-4 LSRL #4

For more information on LSRL, see the *Arm®v8-M Architecture Reference Manual*.

3.6.5 SRSHR

Signed Rounding Shift Right by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =SInt(R[da]);
   op1    =op1 + (1 << (amount-1));
   result =op1 >> amount) [31:0];
   R[da]  =result[31:0];
```

Example operation

If the shift `amount` is 4 and `op1` is `0x87655678`, then:

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =0x87655678;
   op1    =0x87655678 + (1 << (4-1)); // op1 is now 0x87655680, because 1 left shifted
by 3 is 0x8.
   result =(0x87655680 >> 4); // This now behaves as a regular ASR instruction shifted
by 4.
   R[da]  =0xF8765678
```

The following figure shows the regular ASR shift for SRSHR #4.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

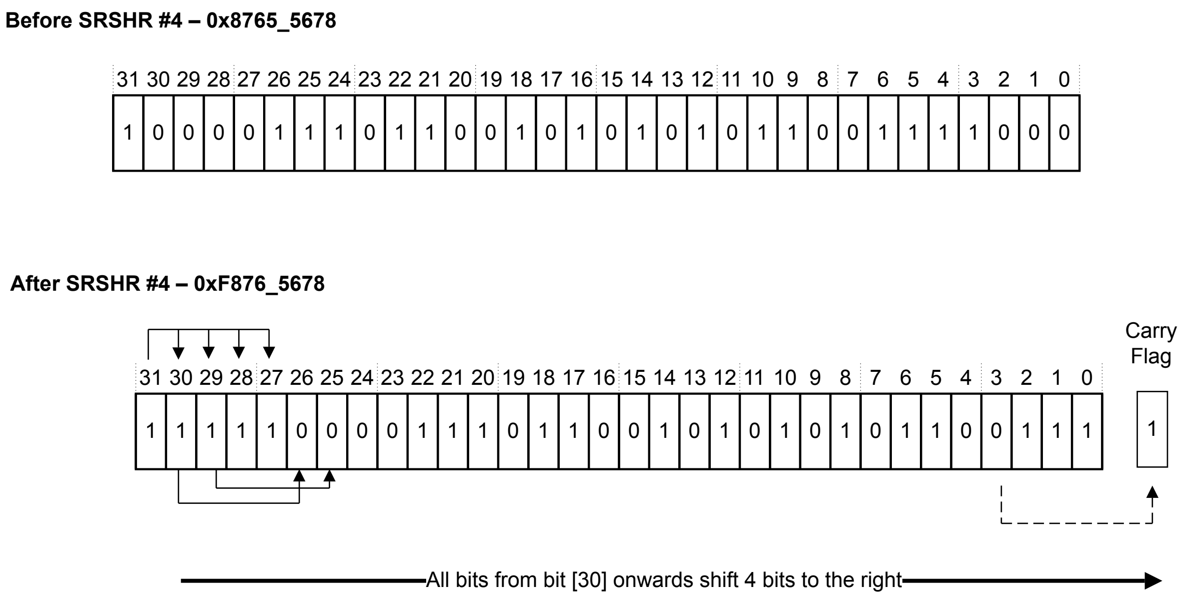


Figure 3-5 SRSHR #4

For more information on SRSHR, see the *Arm®v8-M Architecture Reference Manual*.

3.6.6 SRSHRL

Signed Rounding Shift Right Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =SInt(R[dah]:R[dal]);
   op1    =op1 + (1 << (amount-1));
   result =(op1 >> amount)[63:0];
   R[dah] =result[63:32];
   R[dal] =result[31:0];
```

Example operation

If the shift `amount` is 4 and `R[dah]` and `R[dal]` are `0x87655678` and `0x12345678` respectively, then:

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =(0x87655678:0x12345678);
   op1    =(0x87655678:0x12345678) + (1 << (4-1)); // 0x1 shifted to the left by 3 is
0x8. This is added to R[dah]. Therefore, R[dah] is now 0x87655680 (Rounding operation)
   result =(0x87655680:0x12345678) >> 4)[63:0]; // This now behaves as a regular ASRL
instruction shifted by 4.
   R[dah] =0xF8765567;
   R[dal] =0x01234567;
```

The following figure shows the regular ASRL shift for SRSHRL #4, where `R[dah]` and `R[dal]` are `0x87655678` and `0x12345678` respectively.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

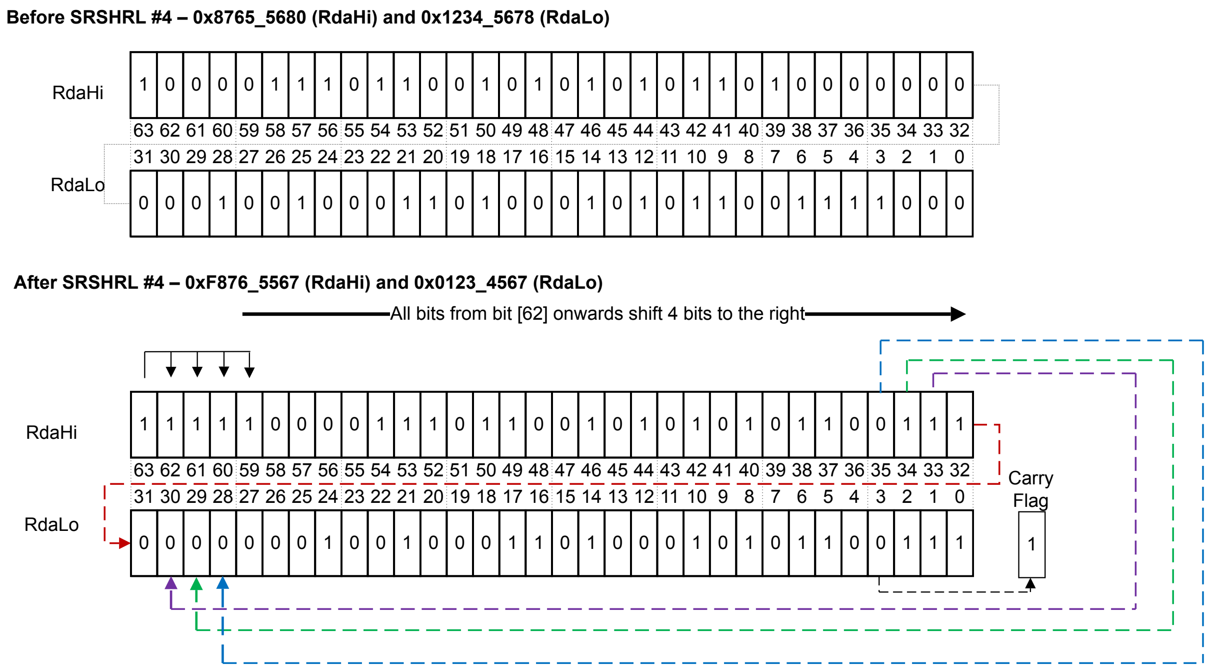


Figure 3-6 SRSHRL #4

For more information on SRSHRL, see the *Arm®v8-M Architecture Reference Manual*.

3.6.7 SQRSHR

Signed Saturating Rounding Shift Right by 0 to 32 bits of a 32-bit value stored in a general-purpose register. If the shift amount is negative, the shift direction is reversed.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount =SInt(R[m][7:0]);
    op1    =SInt(R[da]);
    op1    =op1 + (1 << (amount - 1));
    (result, sat) = SignedSatQ((op1 >> amount), 32);
    if sat then APSR.Q = '1';
    R[da] =result[31:0];
```

Example operation

If the shift `amount` is -4 and `R[da]` is `0x87655678` then:

```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount =-4
    op1    =(0x87655678);
    op1    =0x87655678 + (1 << (-4-1)); // the shift amount is negative, therefore the
shift direction is reversed. This results in op1 remaining as 0x87655678, therefore no
rounding occurs.
    (result, sat) = SignedSatQ((0x87655678 >> -4), 32);
    // Step 1, Sign extension: [63:32] = 0xFFFFFFFF and [31:0] = 0x87655678.
    // Step 2, Shift, left by 4: [63:32] = 0xFFFFFFF8 and [31:0] = 0x76556780.
    // Step 3, saturated = TRUE because 0x76556780 is further away from the maximum
negative value, therefore, [31:0] = 0x800000008:    APSR.Q = '1';
    R[da]  =0x80000000;
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For more information on SQRSHR and the SignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

3.6.8 SQRSHRL

Signed Saturating Rounding Shift Right Long by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. If the shift amount is negative, the shift direction is reversed.

Operation for all encodings

```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount = SInt(R[m][7:0]);
    op1 = SInt(R[dah]:R[dal]);
    op1 = op1 + (1 << (amount - 1));
    (shiftedOp, didSat) = SignedSatQ((op1 >> amount), saturateTo);
    result = SignExtend(shiftedOp, 64);
    if didSat then APSR.Q = '1';
    R[dah] = result[63:32];
    R[dal] = result[31:0];
```

Example operation

If the shift `amount` is -4 and `R[dah]` and `R[dal]` is `0x00008765` and `0x56780000` respectively then:

```
if ConditionPassed() then
    EncodingSpecificOperations();

    amount =-4
    op1    =(0x87650000:0x56780000);
    op1    =(0x87650000:0x56780000) + (1 << (-4-1)); // the shift amount is negative,
therefore the shift direction is reversed. This results in op1 remaining as 0x87655678.
    (shiftedOp, didSat) = SignedSatQ(((0x87650000:0x56780000) >> -4), #48);
    // Step 1, Sign extension: [63:48] = 0xFFFF, [47:16] = 0x87655678, and [15:0] =
0x0000.
    // Step 2, Shift, left by 4: [63:0] = 0xFFFF876556780000.
    // Step 3, saturated = TRUE because 0xFFFF876556780000 is further away from the
maximum negative value.
    result = 0xFFFF800000000000;
    APSR.Q = '1';
     R[dah]  =0xFFFF8000;
     R[dal]  =0x00000000;
```

For more information on SQRSHRL and the SignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

3.6.9 URSHR

Unsigned Rounding Shift Right by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

URSHR behaves the same way as SRSHR, except `op1` can take an unsigned integer value. See *3.6.5 SRSHR* on page 3-111.

3.6.10 URSHRL

Unsigned Rounding Shift Right Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

URSHRL behaves the same way as SRSHRL, except `op1` can take unsigned integer values. See *3.6.6 SRSHRL* on page 3-112.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.7 Left shift operations

Register left shift operations move the bits in a register left by a specified number of bits, the *shift length*.

Register shift can be performed:

- Directly by the shift instructions, and the result is written to a destination register.
- During the calculation of *`Operand2`* by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description for more information. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, *`Rm`* is the register containing the value to be shifted, and *`n`* is the shift length.

3.7.1 LSL

Logical Shift Left shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

The following figure shows an LSL #4 instruction.

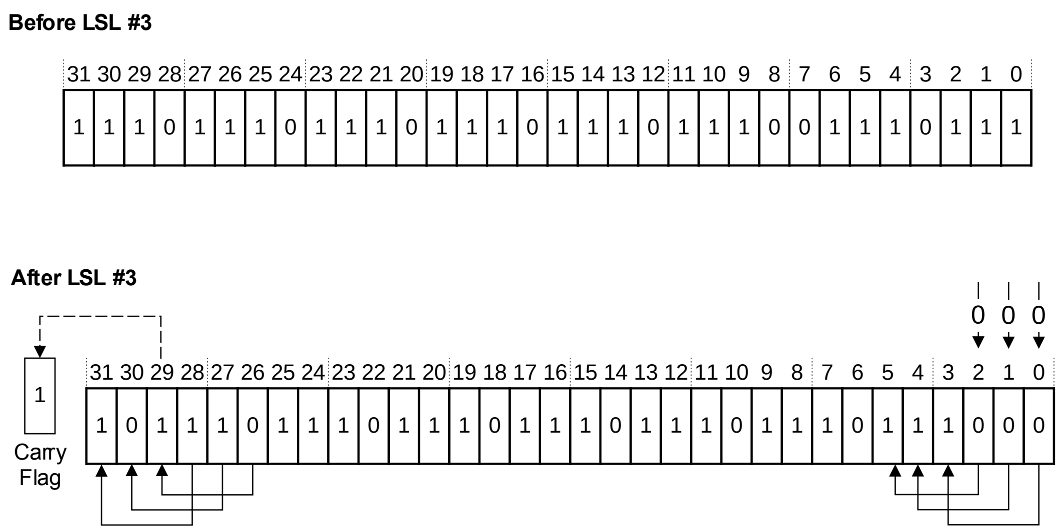


Figure 3-7 LSL #4

For more information on LSL, see the *Arm®v8-M Architecture Reference Manual*.

3.7.2 LSLL

Logical Shift Left Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =UInt(R[dah]:R[dal]);
   op1    =(op1 << amount) [63:0];
   R[dah] =result[63:32];
   R[dal] =result[31:0];
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example operation

If the shift `amount` is 4 and `R[dah]` and `R[dal]` are `0x12345678` and `0x87654321` respectively, then:

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =(0x12345678:0x87654321);
   op1    =((0x12345678:0x87654321) << 4)) [63:0]; //Zeroes are added into bits [63:60]
   R[dah] =0x012345678;
   R[dal] =0x88765432;
```

The following figure shows an LSLL #4 instruction, indicating that zeros are added to the first four MSBs of RdaHi and the carry flag is updated to the last bit shifted out of RdaLo.

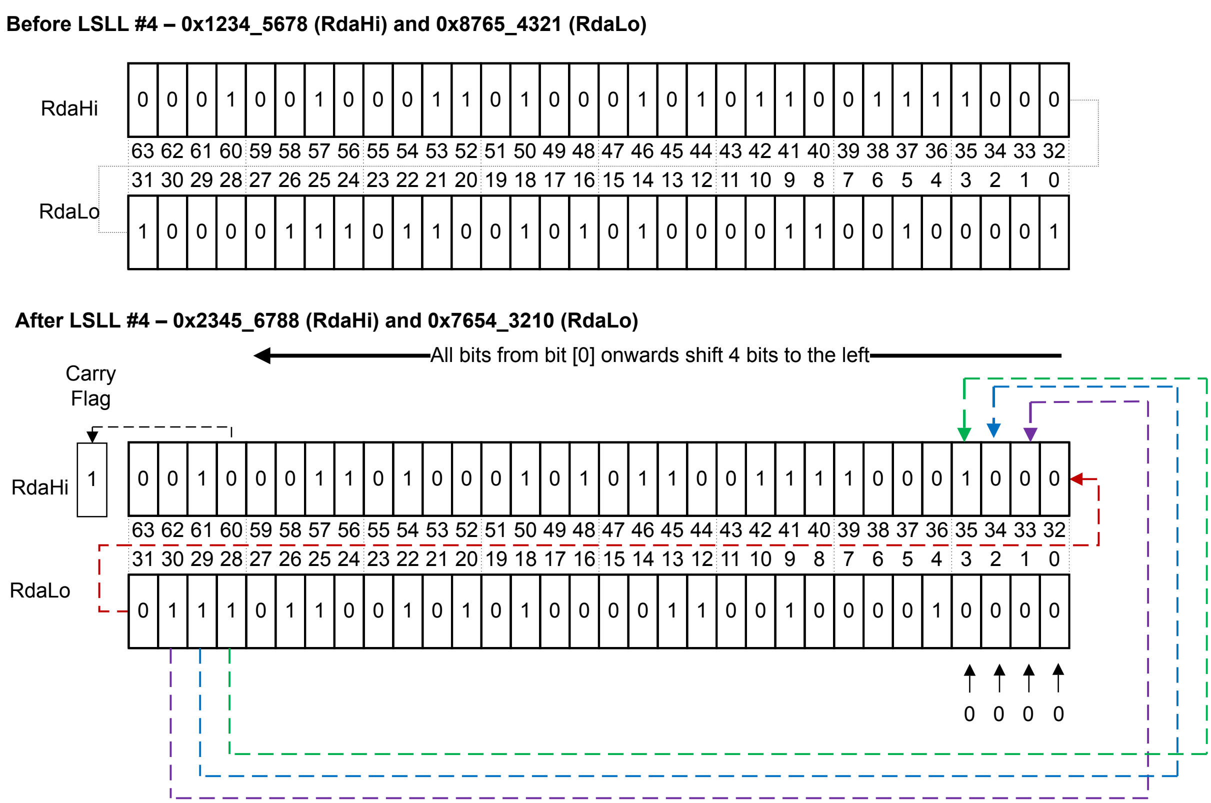


Figure 3-8 LSLL #4

For more information on LSLL, see the *Arm® v8-M Architecture Reference Manual*.

3.7.3 SQSHL

Signed Saturating Shift Left by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

Operation for all encodings

```
if ConditionPassed() then
   EncodingSpecificOperations();

   op1    =SInt(R[da]);
   (result, sat) = SignedSatQ((op1 << amount), 32);
   if sat then APSR.Q = '1';
   R[da] =result[31:0];
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example operation

If the shift `amount` is 4 and `R[da]` is `0x87655678` then:

```
1: if ConditionPassed() then
2:    EncodingSpecificOperations();
3:
4:    op1    =(0x87655678);
5:    (result, sat) = SignedSatQ((0x87655678 << 4), 32);
      // Step 1, Sign extension: [63:32] = 0xFFFFFFFF and [31:0] = 0x87655678.
      // Step 2, Shift, left by 4: [63:32] = 0xFFFFFFFF and [31:0] = 0x76556780.
      // Step 3, saturated = FALSE because 0x76556780 is less than the maximum positive
value. Therefore, [31:0] = 0x765567808:    APSR.Q = '0';
9:    R[da]  =0x76556780;
```

For more information on SQSHL and the SignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

3.7.4 SQSHLL

Signed Saturating Shift Left Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.If the shift amount is negative, the shift direction is reversed.

Operation for all encodings

```
1: if ConditionPassed() then
2:    EncodingSpecificOperations();
3:
4:    op1 = SInt(R[dah]:R[dal]);
5:   (result, sat) = SignedSatQ((op1 << amount), 64);
6:    if sat then APSR.Q = '1';
7:    R[dah] = result[63:32];
8:    R[dal] = result[31:0];
```

Example operation

If the shift `amount` is 4 and `R[dah]` and `R[dal]` are `0x87655678` and `0x12345678` respectively, then:

```
1: if ConditionPassed() then
2:    EncodingSpecificOperations();
3:
4:    op1    =(0x87655678:0x12345678);
5:    (result, sat) = SignedSatQ((0x8765567812345678 << 4), 64);
      // Step 1, Sign extension: [128:64] = 0xFFFFFFFFFFFF, [63:32] = 0x87655678 and
[31:0] = 0x12345678.
      // Step 2, Shift, left by 4: [63:32] = 0x76556781 and [31:0] = 0x23456780.
      // Step 3, saturated = TRUE because 0x7655678123456780 is less that the maximum
positive value. Therefore, [64:0] = 0x7FFFFFFFFFFFFFF8:    APSR.Q = '0';
9:    R[da]  =0x7655678123456780;
```

For more information on SQSHLL and the SignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

3.7.5 UQSHL

Unsigned Saturating Shift Left by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

UQSHL behaves the same way as SQSHL, except `op1` can take an unsigned integer value. See *3.6.5 SRSHR* on page 3-111.

3.7.6 UQSHLL

Unsigned Saturating Shift Left Long by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

UQSHLL behaves the same way as SQSHLL, except `op1` can take unsigned integer values. See *3.6.5 SRSHR* on page 3-111.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.7.7 UQRSHL

Unsigned Saturating Rounding Shift Left. Unsigned saturating rounding shift left by 0 to 32 bits of a 32-bit value stored in a general-purpose register. If the shift amount is negative, the shift direction is reversed.

UQRSHL behaves the same way as SQRSHR, except `op1` can take an unsigned integer value. See *3.6.5 SRSHR* on page 3-111.

For more information on UQRSHL and the UnsignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

3.7.8 UQRSHLL

Unsigned Saturating Rounding Shift Left Long. Unsigned saturating rounding shift left by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. If the shift amount is negative, the shift direction is reversed.

UQRSHLL behaves the same way as SQRSHRL, except `op1` can take unsigned integer values. See *3.6.5 SRSHR* on page 3-111.

For more information on UQRSHLL and the UnsignedSatQ function, see the *Arm®v8-M Architecture Reference Manual*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.8 Rotate shift operations

Register rotate shift operations rotate the bits in a register by a specified number of bits, the *shift length*.

Register shift can be performed:

- Directly by the shift instructions, and the result is written to a destination register.
- During the calculation of `Operand2` by the instructions that specify the second operand as a register with shift. The result is used by the instruction.

The permitted shift lengths depend on the shift type and the instruction, see the individual instruction description for more information. If the shift length is 0, no shift occurs. Register shift operations update the carry flag except when the specified shift length is 0. The following sub-sections describe the various shift operations and how they affect the carry flag. In these descriptions, *Rm* is the register containing the value to be shifted, and *n* is the shift length.

Note

Some of the shift operation descriptions in this section do not have example diagrams illustrating the operation. These will be added for the next release.

3.8.1 ROR

Rotate Right (register). Rotate Right (register) rotates a register value by a variable number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. This instruction is an alias of the MOV, MOVS (register-shifted register) instruction.

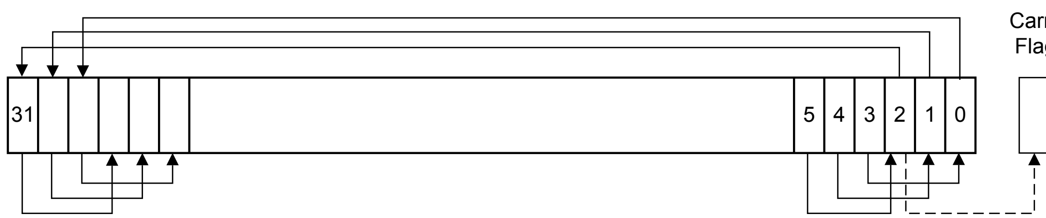


Figure 3-9 ROR #3

3.8.2 RORS

Rotate Right, Setting flags (immediate). Rotate Right, Setting flags (immediate) rotates a register value by a constant number of bits, inserting the bits that are rotated off the right end into the vacated bit positions on the left, writes the result to the destination register, and updates the condition flags based on the result.

3.8.3 RRX

Rotate Right with Extend. Rotate Right with Extend shifts a register value right by one bit, shifting the Carry flag into bit[31], and writes the result to the destination register. This instruction is an alias of the MOV (register) instruction.

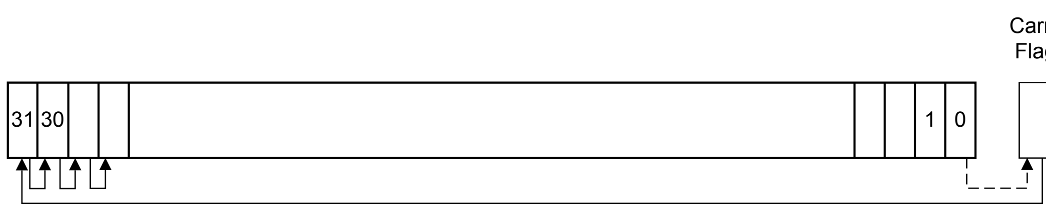


Figure 3-10 RRX

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.8.4 RRXS

Rotate Right with Extend, Setting flags. Rotate Right with Extend, Setting flags shifts a register value right by one bit, shifting the Carry flag into bit[31] and bit[0] into the Carry flag, writes the result to the destination register and updates the condition flags (other than Carry) based on the result.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.9 Address alignment

An aligned access is an operation where a word-aligned address is used for a word, dual word, or multiple word access, or where a halfword-aligned address is used for a halfword access. Byte accesses are always aligned.

The Cortex-M55 processor supports unaligned access only for the following instructions:

- `LDR`, `LDRT`.
- `LDRH`, `LDRHT`.
- `LDRSH`, `LDRSHT`.
- `STR`, `STRT`.
- `STRH`, `STRHT`.

All other load and store instructions generate a UsageFault exception if they perform an unaligned access, and therefore their accesses must be address aligned.

Unaligned accesses are usually slower than aligned accesses. In addition, some memory regions might not support unaligned accesses. Therefore, Arm recommends that programmers ensure that accesses are aligned. To trap accidental generation of unaligned accesses, use the UNALIGN_TRP bit in the Configuration and Control Register.

MVE instructions operate on a fixed vector width of 128 bits, but their alignment requirements are dependent on the element size.

The permitted lane widths and lane operations per beat, are:

- For a 64-bit lane size, a beat performs half of the lane operation.
- For a 32-bit lane size, a beat performs a one lane operation.
- For a 16-bit lane size, a beat performs a two lane operations.
- For an 8-bit lane size, a beat performs a four lane operations.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.10 PCrelative expressions

A PC--relative expression or *label* is a symbol that represents the address of an instruction or literal data. It is represented in the instruction as the PC value plus or minus a numeric offset. The assembler calculates the required offset from the label and the address of the current instruction. If the offset is too big, the assembler produces an error.

Note

- For `B`, `BL`, `CBNZ`, and `CBZ` instructions, the value of the PC is the address of the current instruction plus 4 bytes.
- For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.
- Your assembler might permit other syntaxes for PC-relative expressions, such as a label plus or minus a number, or an expression of the form `[PC, #number]`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.11 Conditional execution

Most data processing instructions can optionally update the condition flags in the *Application Program Status Register* (APSR) according to the result of the operation. Some instructions update all flags, and some only update a subset. If a flag is not updated, the original value is preserved. See the instruction descriptions for the flags they affect.

You can execute an instruction conditionally, based on the condition flags set in another instruction, either:

- Immediately after the instruction that updated the flags.
- After any number of intervening instructions that have not updated the flags.

Conditional execution is available by using conditional branches or by adding condition code suffixes to instructions. The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- Does not execute.
- Does not write any value to its destination register.
- Does not affect any of the flags.
- Does not generate any exception.

Conditional instructions, except for conditional branches, must be inside an If-Then instruction block. Depending on the vendor, the assembler might automatically insert an `IT` instruction if you have conditional instructions outside the IT block.

Use the `CBZ` and `CBNZ` instructions to compare the value of a register against zero and branch on the result.

This section contains the following subsections:

- *3.11.1 The condition flags* on page 3-124.
- *3.11.2 Condition code suffixes* on page 3-124.
- *3.11.3 Predication* on page 3-125.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.11.1 The condition flags

The APSR contains the N, Z, C, and V condition flags.

N Set to 1 when the result of the operation was negative, cleared to 0 otherwise.

Z Set to 1 when the result of the operation was zero, cleared to 0 otherwise.

C Set to 1 when the operation resulted in a carry, cleared to 0 otherwise.

V Set to 1 when the operation caused overflow, cleared to 0 otherwise.

The C condition flag is set in one of four ways:

- For an addition, including the comparison instruction `CMN`, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
- For a subtraction, including the comparison instruction `CMP`, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition or subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition or subtractions, C is normally left unchanged. See the individual instruction descriptions for any special cases.

Overflow occurs when the sign of the result, in bit[31], does not match the sign of the result had the operation been performed at infinite precision. For example, the V condition flag can be set in one of four ways:

- If adding two negative values results in a positive value.
- If adding two positive values results in a negative value.
- If subtracting a positive value from a negative value generates a positive value.
- If subtracting a negative value from a positive value generates a negative value.

The Compare operations are identical to subtracting, for `CMP`, or adding, for `CMN`, except that the result is discarded. See the instruction descriptions for more information.

Note

Most instructions update the status flags only if the S suffix is specified. See the instruction descriptions for more information.

3.11.2 Condition code suffixes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as {*cond*}. Conditional execution requires a preceding `IT` instruction. An instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition.

You can use conditional execution with the `IT` instruction to reduce the number of branch instructions in code.

The following table also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 3-5 Condition code suffixes

Suffix	Flags	Meaning
`EQ`	Z = 1	Equal.
`NE`	Z = 0	Not equal.
`CS or HS`	C = 1	Higher or same, unsigned.
`CC or LO`	C = 0	Lower, unsigned.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-5 Condition code suffixes (continued)

Suffix	Flags	Meaning
MI	N = 1	Negative.
PL	N = 0	Positive or zero.
VS	V = 1	Overflow.
VC	V = 0	No overflow.
HI	C = 1 and Z = 0	Higher, unsigned.
LS	C = 0 or Z = 1	Lower or same, unsigned.
GE	N = V	Greater than or equal, signed.
LT	N != V	Less than, signed.
GT	Z = 0 and N = V	Greater than, signed.
LE	Z = 1 and N != V	Less than or equal, signed.
AL	Can have any value	Always. This is the default when no suffix is specified.

The following example shows the use of a conditional instruction to find the absolute value of a number. `R0` = `abs(R1)`.

Absolute value

```
MOVS    R0, R1          ; R0 = R1, setting flags.
IT      MI              ; Skipping next instruction if value 0 or positive.
RSBMI   R0, R0, #0      ; If negative, R0 = -R0.
```

The following example shows the use of conditional instructions to update the value of `R4` if the signed values `R0` is greater than `R1` and `R2` is greater than `R3`.

Compare and update value

```
CMP     R0, R1      ; Compare R0 and R1, setting flags.
ITT     GT          ; Skip next two instructions unless GT condition holds.
CMPGT   R2, R3      ; If 'greater than', compare R2 and R3, setting flags.
MOVGT   R4, R5      ; If still 'greater than', do R4 = R5.
```

3.11.3 Predication

MVE includes predication that enables the independent masking of each lane within a vector operation.

It supports the following predication mechanisms:

Loop tail predication
This eliminates the requirement for special vector tail handling code after loops where the number of elements to be processed is not a multiple of the number of elements in the vector.

VPT predication
This enables data-dependent conditions that are based on data value comparisons to mask

Loop tail predication and VPT predication operate separately. The resulting predication flags from each mechanism are ANDed together so that a lane of a vector operation is only active if both the loop tail predication and the VPT predication conditions are true.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.12 Instruction width selection

There are many instructions that can generate either a 16-bit encoding or a 32-bit encoding depending on the operands and destination register specified. For some of these instructions, you can force a specific instruction size by using an instruction width suffix. The `.W` suffix forces a 32-bit instruction encoding. The `.N` suffix forces a 16-bit instruction encoding.

If you specify an instruction width suffix and the assembler cannot generate an instruction encoding of the requested width, it generates an error.

Note

In some cases it might be necessary to specify the `.W` suffix, for example if the operand is the label of an instruction or literal data, as in the case of branch instructions. This is because the assembler might not automatically generate the right size encoding.

To use an instruction width suffix, place it immediately after the instruction mnemonic and condition code, if any. The following example shows instructions with the instruction width suffix.

Instruction width selection

```
BCS.W  label      ; Creates a 32-bit instruction even for a short branch.
ADDS.W R0, R0, R1 ; Creates a 32-bit instruction even though the same
                  ; operation can be done by a 16-bit instruction.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13 General data processing instructions

Reference material for the Cortex-M55 processor data processing instruction set.

3.13.1 List of data processing instructions

An alphabetically ordered list of the data processing instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-6 Data processing instructions

Mnemonic	Brief description	See
`ADC`	Add with Carry	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`ADD`	Add	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`ADDW`	Add	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`AND`	Logical AND	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`ASR`	Arithmetic Shift Right	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`BIC`	Bit Clear	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`CLZ`	Count leading zeros	*3.13.5 CLZ* on page 3-134
`CMN`	Compare Negative	*3.13.6 CMP and CMN* on page 3-135
`CMP`	Compare	*3.13.6 CMP and CMN* on page 3-135
`EOR`	Exclusive OR	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`LSL`	Logical Shift Left	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`LSR`	Logical Shift Right	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`MOV`	Move	*3.13.7 MOV and MVN* on page 3-136
`MOVT`	Move Top	*3.13.8 MOVT* on page 3-138
`MOVW`	Move 16-bit constant	*3.13.7 MOV and MVN* on page 3-136
`MVN`	Move NOT	*3.13.7 MOV and MVN* on page 3-136
`ORN`	Logical OR NOT	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`ORR`	Logical OR	*3.13.3 AND, ORR, EOR, BIC, and ORN* on page 3-131
`RBIT`	Reverse Bits	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
`REV`	Reverse byte order in a word	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
`REV16`	Reverse byte order in each halfword	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
`REVSH`	Reverse byte order in bottom halfword and sign extend	*3.13.9 REV, REV16, REVSH, and RBIT* on page 3-139
`ROR`	Rotate Right	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`RRX`	Rotate Right with Extend	*3.13.4 ASR, LSL, LSR, ROR, and RRX* on page 3-132
`RSB`	Reverse Subtract	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
`SADD16`	Signed Add 16	*3.13.10 SADD16 and SADD8* on page 3-140
`SADD8`	Signed Add 8	*3.13.10 SADD16 and SADD8* on page 3-140
`SASX`	Signed Add and Subtract with Exchange	*3.13.11 SASX and SSAX* on page 3-142
`SEL`	Select bytes	*3.13.12 SEL* on page 3-144

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-6 Data processing instructions (continued)

Mnemonic	Brief description	See
SSAX	Signed Subtract and Add with Exchange	*3.13.11 SASX and SSAX* on page 3-142
SBC	Subtract with Carry	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
SHADD16	Signed Halving Add 16	*3.13.13 SHADD16 and SHADD8* on page 3-145
SHADD8	Signed Halving Add 8	*3.13.13 SHADD16 and SHADD8* on page 3-145
SHASX	Signed Halving Add and Subtract with Exchange	*3.13.14 SHASX and SHSAX* on page 3-146
SHSAX	Signed Halving Subtract and Add with Exchange	*3.13.14 SHASX and SHSAX* on page 3-146
SHSUB16	Signed Halving Subtract 16	*3.13.15 SHSUB16 and SHSUB8* on page 3-147
SHSUB8	Signed Halving Subtract 8	*3.13.15 SHSUB16 and SHSUB8* on page 3-147
SSUB16	Signed Subtract 16	*3.13.16 SSUB16 and SSUB8* on page 3-148
SSUB8	Signed Subtract 8	*3.13.16 SSUB16 and SSUB8* on page 3-148
SUB	Subtract	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
SUBW	Subtract	*3.13.2 ADD, ADC, SUB, SBC, and RSB* on page 3-129
TEQ	Test Equivalence	*3.13.17 TST and TEQ* on page 3-150
TST	Test	*3.13.17 TST and TEQ* on page 3-150
UADD16	Unsigned Add 16	*3.13.18 UADD16 and UADD8* on page 3-151
UADD8	Unsigned Add 8	*3.13.18 UADD16 and UADD8* on page 3-151
UASX	Unsigned Add and Subtract with Exchange	*3.13.19 UASX and USAX* on page 3-153
USAX	Unsigned Subtract and Add with Exchange	*3.13.19 UASX and USAX* on page 3-153
UHADD16	Unsigned Halving Add 16	*3.13.20 UHADD16 and UHADD8* on page 3-155
UHADD8	Unsigned Halving Add 8	*3.13.20 UHADD16 and UHADD8* on page 3-155
UHASX	Unsigned Halving Add and Subtract with Exchange	*3.13.21 UHASX and UHSAX* on page 3-156
UHSAX	Unsigned Halving Subtract and Add with Exchange	*3.13.21 UHASX and UHSAX* on page 3-156
UHSUB16	Unsigned Halving Subtract 16	*3.13.22 UHSUB16 and UHSUB8* on page 3-158
UHSUB8	Unsigned Halving Subtract 8	*3.13.22 UHSUB16 and UHSUB8* on page 3-158
USAD8	Unsigned Sum of Absolute Differences	*3.13.23 USAD8* on page 3-159
USADA8	Unsigned Sum of Absolute Differences and Accumulate	*3.13.24 USADA8* on page 3-160
USUB16	Unsigned Subtract 16	*3.13.25 USUB16 and USUB8* on page 3-161
USUB8	Unsigned Subtract 8	*3.13.25 USUB16 and USUB8* on page 3-161

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.2 ADD, ADC, SUB, SBC, and RSB

Add, Add with carry, Subtract, Subtract with carry, and Reverse Subtract.

Syntax

`op{S}{cond} {Rd,} Rn, Operand2 ; ADD; ADC; SBC; RSB`

`op{S|W}{cond} {Rd,} Rn, #imm12 ; ADD; SUB`

Where:

op	Is one of:
	`ADD` Add.
	`ADC` Add with Carry.
	`SUB` Subtract.
	`SBC` Subtract with Carry.
	`RSB` Reverse Subtract.
`S`	Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn	Is the register holding the first operand.
Operand2	Is a flexible second operand.
imm12	Is any value in the range 0-4095.

Operation

The `ADD` instruction adds the value of *Operand2* or *imm12* to the value in *Rn*.

The `ADC` instruction adds the values in *Rn* and *Operand2*, together with the carry flag.

The `SUB` instruction subtracts the value of *Operand2* or *imm12* from the value in *Rn*.

The `SBC` instruction subtracts the value of *Operand2* from the value in *Rn*. If the carry flag is clear, the result is reduced by one.

The `RSB` instruction subtracts the value in *Rn* from the value of *Operand2*. This is useful because of the wide range of options for *Operand2*.

Use `ADC` and `SBC` to synthesize multiword arithmetic.

Note

`ADDW` is equivalent to the `ADD` syntax that uses the *imm12* operand. `SUBW` is equivalent to the `SUB` syntax that uses the *imm12* operand.

Restrictions

In these instructions:

- *Operand2* must not be SP and must not be PC.
- *Rd* can be SP only in `ADD` and `SUB`, and only with the additional restrictions:
  - *Rn* must also be SP.
  - Any shift in *Operand2* must be limited to a maximum of 3 bits using `LSL`.
- *Rn* can be SP only in `ADD` and `SUB`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- `Rd` can be PC only in the `ADD{cond} PC, PC, Rm` instruction where:
  - — You must not specify the S suffix.
  - — `Rm` must not be PC and must not be SP.
  - — If the instruction is conditional, it must be the last instruction in the IT block.
- with the exception of the `ADD{cond} PC, PC, Rm` instruction, `Rn` can be PC only in `ADD` and `SUB`, and only with the additional restrictions:
  - — You must not specify the S suffix.
  - — The second operand must be a constant in the range 0-4095.

Note

— When using the PC for an addition or a subtraction, bits[1:0] of the PC are rounded to `0b00` before performing the calculation, making the base address for the calculation word-aligned.

— If you want to generate the address of an instruction, you have to adjust the constant based on the value of the PC. Arm recommends that you use the `ADR` instruction instead of `ADD` or `SUB` with `Rn` equal to the PC, because your assembler automatically calculates the correct constant for the `ADR` instruction.

When `Rd` is PC in the `ADD{cond} PC, PC, Rm` instruction:
- Bit[0] of the value written to the PC is ignored.
- A branch occurs to the address created by forcing bit[0] of that value to 0.

Condition flags

If `S` is specified, these instructions update the N, Z, C and V flags according to the result.

Example 3-1 Examples

```
ADD     R2, R1, R3
SUBS    R8, R6, #240        ; Sets the flags on the result.
RSB     R4, R4, #1280       ; Subtracts contents of R4 from 1280.
ADCHI   R11, R0, R3         ; Only executed if C flag set and Z.
                            ; flag clear.
```

Multiword arithmetic examples

The following example shows two instructions that add a 64-bit integer contained in `R2` and `R3` to another 64-bit integer contained in `R0` and `R1`, and place the result in `R4` and `R5`.

64-bit addition

```
ADDS    R4, R0, R2    ; Add the least significant words.
ADC     R5, R1, R3    ; Add the most significant words with
carry.
```

Multiword values do not have to use consecutive registers. The following example shows instructions that subtract a 96-bit integer contained in R9, R1, and R11 from another contained in R6, R2, and R8. The example stores the result in R6, R9, and R2.

96-bit subtraction

```
SUBS    R6, R6, R9     ; Subtract the least significant words.
SBCS    R9, R2, R1     ; Subtract the middle words with carry.
SBC     R2, R8, R11    ; Subtract the most significant words
with carry.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.3 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Syntax

`op{S}{cond} {Rd,} Rn, Operand2`

Where:

op
Is one of:

`AND`	Logical AND.
`ORR`	Logical OR, or bit set.
`EOR`	Logical Exclusive OR.
`BIC`	Logical AND NOT, or bit clear.
`ORN`	Logical OR NOT.

`S`
Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.

cond
Is an optional condition code.

Rd
Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn
Is the register holding the first operand.

Operand2
Is a flexible second operand.

Operation

The `AND`, `EOR`, and `ORR` instructions perform bitwise AND, Exclusive OR, and OR operations on the values in *Rn* and *Operand2*.

The `BIC` instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

The `ORN` instruction performs an OR operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

Restrictions

Do not use SP and do not use PC.

Condition flags

If `S` is specified, these instructions:
- Update the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*.
- Do not affect the V flag.

Example 3-2 Examples

```
AND     R9, R2, #0xFF00
ORREQ   R2, R0, R5
ANDS    R9, R8, #0x19
EORS    R7, R11, #0x18181818
BIC     R0, R1, #0xab
ORN     R7, R11, R14, ROR #4
ORNS    R7, R11, R14, ASR #32
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.4 ASR, LSL, LSR, ROR, and RRX

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, Rotate Right, and Rotate Right with Extend.

Syntax

`op{S}{cond} Rd, Rm, Rs`

`op{S}{cond} Rd, Rm, #n`

`RRX{S}{cond} Rd, Rm`

Where:

op Is one of:

`ASR`	Arithmetic Shift Right.
`LSL`	Logical Shift Left.
`LSR`	Logical Shift Right.
`ROR`	Rotate Right.

S Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.

Rd Is the destination register.

Rm Is the register holding the value to be shifted.

Rs Is the register holding the shift length to apply to the value in *Rm*. Only the least significant byte is used and can be in the range 0-255.

n Is the shift length. The range of shift length depends on the instruction:

`ASR`	Shift length from 1 to 32
`LSL`	Shift length from 0 to 31
`LSR`	Shift length from 1 to 32
`ROR`	Shift length from 1 to 31.

Note

`MOVS` *Rd, Rm* is the preferred syntax for `LSLS` *Rd, Rm, #0*.

Operation

`ASR`, `LSL`, `LSR`, and `ROR` move the bits in the register *Rm* to the left or right by the number of places specified by constant *n* or register *Rs*.

`RRX` moves the bits in register *Rm* to the right by 1.

In all these instructions, the result is written to *Rd*, but the value in register *Rm* remains unchanged. For details on what result is generated by the different instructions.

Restrictions

Do not use SP and do not use PC.

Condition flags

If `S` is specified:

- These instructions update the N, Z and C flags according to the result.
- The C flag is updated to the last bit shifted out, except when the shift length is 0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-3 Examples

```
ASR    R7, R8, #9  ; Arithmetic shift right by 9 bits.
LSLS   R1, R2, #3  ; Logical shift left by 3 bits with flag update.
LSR    R4, R5, #6  ; Logical shift right by 6 bits.
ROR    R4, R5, R6  ; Rotate right by the value in the bottom byte of R6.
RRX    R4, R5      ; Rotate right with extend.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.5 CLZ

Count Leading Zeros.

Syntax

`CLZ{cond} Rd, Rm`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
Rm	Is the operand register.

Operation

The `CLZ` instruction counts the number of leading zeros in the value in *Rm* and returns the result in *Rd*. The result value is 32 if no bits are set and zero if bit[31] is set.

Restrictions

Do not use SP and do not use PC.

Condition flags

This instruction does not change the flags.

Example 3-4 Examples

```
CLZ     R4,R9
CLZNE   R2,R3
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.6 CMP and CMN

Compare and Compare Negative.

Syntax

`CMP{cond} Rn, Operand2`

`CMN{cond} Rn, Operand2`

Where:

cond	Is an optional condition code.
Rn	Is the register holding the first operand.
Operand2	Is a flexible second operand.

Operation

These instructions compare the value in a register with *Operand2*. They update the condition flags on the result, but do not write the result to a register.

The `CMP` instruction subtracts the value of *Operand2* from the value in *Rn*. This is the same as a `SUBS` instruction, except that the result is discarded.

The `CMN` instruction adds the value of *Operand2* to the value in *Rn*. This is the same as an `ADDS` instruction, except that the result is discarded.

Restrictions

In these instructions:
- Do not use PC.
- *Operand2* must not be SP.

Condition flags

These instructions update the N, Z, C and V flags according to the result.

Example 3-5 Examples

```
CMP     R2, R9
CMN     R0, #6400
CMPGT   SP, R7, LSL #2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.7 MOV and MVN

Move and Move NOT.

Syntax

`MOV{S}{cond} Rd, Operand2`

`MOV{S}{cond} Rd, Rm`

`MOV{W}{cond} Rd, #imm16`

`MVN{S}{cond} Rd, Operand2`

Where:

`S`	Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
cond	Is an optional condition code.
Rd	Is the destination register.
Operand2	Is a flexible second operand.
Rm	The source register.
imm16	Is any value in the range 0-65535.

Operation

The `MOV` instruction copies the value of *Operand2* into *Rd*.

When *Operand2* in a `MOV` instruction is a register with a shift other than `LSL #0`, the preferred syntax is the corresponding shift instruction:Also, the `MOV` instruction permits additional forms of *Operand2* as synonyms for shift instructions:

- `ASR{S}{cond} Rd, Rm, #n` is the preferred syntax for `MOV{S}{cond} Rd, Rm, ASR #n`.
- `LSL{S}{cond} Rd, Rm, #n` is the preferred syntax for `MOV{S}{cond} Rd, Rm, LSL #n` if *n* != 0.
- `LSR{S}{cond} Rd, Rm, #n` is the preferred syntax for `MOV{S}{cond} Rd, Rm, LSR #n`.
- `ROR{S}{cond} Rd, Rm, #n` is the preferred syntax for `MOV{S}{cond} Rd, Rm, ROR #n`.
- `RRX{S}{cond} Rd, Rm` is the preferred syntax for `MOV{S}{cond} Rd, Rm, RRX`.

- `MOV{S}{cond} Rd, Rm, ASR Rs` is a synonym for `ASR{S}{cond} Rd, Rm, Rs`.
- `MOV{S}{cond} Rd, Rm, LSL Rs` is a synonym for `LSL{S}{cond} Rd, Rm, Rs`.
- `MOV{S}{cond} Rd, Rm, LSR Rs` is a synonym for `LSR{S}{cond} Rd, Rm, Rs`.
- `MOV{S}{cond} Rd, Rm, ROR Rs` is a synonym for `ROR{S}{cond} Rd, Rm, Rs`.

The `MVN` instruction takes the value of *Operand2*, performs a bitwise logical NOT operation on the value, and places the result into *Rd*.

Note

The `MOVW` instruction provides the same function as `MOV`, but is restricted to using the *imm16* operand.

Restrictions

You can use SP and PC only in the `MOV` instruction, with the following restrictions:

- The second operand must be a register without shift.
- You must not specify the `S` suffix.

When *Rd* is PC in a `MOV` instruction:
- Bit[0] of the value written to the PC is ignored.
- A branch occurs to the address created by forcing bit[0] of that value to 0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

Though it is possible to use `MOV` as a branch instruction, Arm strongly recommends the use of a `BX` or `BLX` instruction to branch for software portability to the Arm instruction set.

Condition flags

If `S` is specified, these instructions:

- Update the N and Z flags according to the result.
- Can update the C flag during the calculation of *`Operand2`*.
- Do not affect the V flag.

Example 3-6 Examples

```
MOVS  R11, #0x000B    ; Write value of 0x000B to R11, flags get updated.
MOV   R1, #0xFA05     ; Write value of 0xFA05 to R1, flags are not updated.
MOVS  R10, R12        ; Write value in R12 to R10, flags get updated.
MOV   R3, #23         ; Write value of 23 to R3.
MOV   R8, SP          ; Write value of stack pointer to R8.
MVNS  R2, #0xF        ; Write value of 0xFFFFFFF0 (bitwise inverse of 0xF).
                      ; to the R2 and update flags.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.8 MOVT

Move Top.

Syntax

`MOVT{cond} Rd, #imm16`

Where:

cond Is an optional condition code.

Rd Is the destination register.

imm16 Is a 16-bit immediate constant and must be in the range 0-65535.

Operation

`MOVT` writes a 16-bit immediate value, *imm16*, to the top halfword, *Rd*[31:16], of its destination register. The write does not affect *Rd*[15:0].

The `MOV`, `MOVT` instruction pair enables you to generate any 32-bit constant.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Example 3-7 Examples

```
MOVT   R3, #0xF123 ; Write 0xF123 to upper halfword of R3, lower halfword
                   ; and APSR are unchanged.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.9 REV, REV16, REVSH, and RBIT

Reverse bytes and Reverse bits.

Syntax

op{cond} Rd, Rn

Where:

op
Is one of:

`REV`	Reverse byte order in a word.
`REV16`	Reverse byte order in each halfword independently.
`REVSH`	Reverse byte order in the bottom halfword, and sign extend to 32 bits.
`RBIT`	Reverse the bit order in a 32-bit word.

cond
Is an optional condition code.

Rd
Is the destination register.

Rn
Is the register holding the operand.

Operation

Use these instructions to change endianness of data:

`REV`

converts either:
- 32-bit big-endian data into little-endian data.
- 32-bit little-endian data into big-endian data.

`REV16`

converts either:
- 16-bit big-endian data into little-endian data.
- 16-bit little-endian data into big-endian data.

`REVSH`

converts either:
- 16-bit signed big-endian data into 32-bit signed little-endian data.
- 16-bit signed little-endian data into 32-bit signed big-endian data.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-8 Examples

```
REV    R3, R7  ; Reverse byte order of value in R7 and write it to R3.
REV16  R0, R0  ; Reverse byte order of each 16-bit halfword in R0.
REVSH  R0, R5  ; Reverse Signed Halfword.
REVHS  R3, R7  ; Reverse with Higher or Same condition.
RBIT   R7, R8  ; Reverse bit order of value in R8 and write the result to R7.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.10 SADD16 and SADD8

Signed Add 16 and Signed Add 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	`SADD16`	Performs two 16-bit signed integer additions.
	`SADD8`	Performs four 8-bit signed integer additions.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to perform a halfword or byte add in parallel.

The `SADD16` instruction:The `SADD8` instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Writes the result in the corresponding halfwords of the destination register.

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Writes the result in the corresponding bytes of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results of the additions.

For `SADD16`:

```
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]<15:0> = sum1<15:0>;
   R[d]<31:16> = sum2<15:0>;
   APSR.GE<1:0> = if sum1 >= 0 then '11' else '00';
   APSR.GE<3:2> = if sum2 >= 0 then '11' else '00';
```

For `SADD8`:

```
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]<7:0> = sum1<7:0>;
   R[d]<15:8> = sum2<7:0>;
   R[d]<23:16> = sum3<7:0>;
   R[d]<31:24> = sum4<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';
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-9 Examples

```
SADD16 R1, R0      ; Adds the halfwords in R0 to the corresponding halfwords of
                   ; R1 and writes to corresponding halfword of R1.SADD8  R4, R0, R5
                   ; Adds bytes of R0 to the corresponding byte in R5 and writes
                   ; to the corresponding byte in R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.11 SASX and SSAX

Signed Add and Subtract with Exchange and Signed Subtract and Add with Exchange.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	SASX	Signed Add and Subtract with Exchange.
	SSAX	Signed Subtract and Add with Exchange.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

The SASX instruction:

1. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
2. Writes the signed result of the addition to the top halfword of the destination register.
3. Subtracts the signed bottom halfword of the second operand from the top signed halfword of the first operand.
4. Writes the signed result of the subtraction to the bottom halfword of the destination register.

The SSAX instruction:

1. Subtracts the signed bottom halfword of the second operand from the top signed halfword of the first operand.
2. Writes the signed result of the addition to the bottom halfword of the destination register.
3. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
4. Writes the signed result of the subtraction to the top halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results.

For SASX:

```
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]<15:0> = diff<15:0>;
   R[d]<31:16> = sum<15:0>;
   APSR.GE<1:0> = if diff >= 0 then '11' else '00';
   APSR.GE<3:2> = if sum >= 0 then '11' else '00';
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

For SSAX:

```
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]<15:0> = sum<15:0>;
   R[d]<31:16> = diff<15:0>;
   APSR.GE<1:0> = if sum >= 0 then '11' else '00';
   APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```

Example 3-10 Examples

```
SASX    R0, R4, R5     ; Adds top halfword of R4 to bottom halfword of R5 and
                       ; writes to top halfword of R0.
                       ; Subtracts bottom halfword of R5 from top halfword of R4
                       ; and writes to bottom halfword of R0.
SSAX    R7, R3, R2     ; Subtracts top halfword of R2 from bottom halfword of R3
                       ; and writes to bottom halfword of R7.
                       ; Adds top halfword of R3 with bottom halfword of R2 and
                       ; writes to top halfword of R7.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.12 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.

Syntax

`SEL{cond} {Rd,} Rn, Rm`

Where:

cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn	Is the first operand register.
Rm	Is the second operand register.

Operation

The `SEL` instruction:

1. Reads the value of each bit of APSR.GE.
2. Depending on the value of APSR.GE, assigns the destination register the value of either the first or second operand register.

The behavior is:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    R[d]<7:0> = if APSR.GE<0> == '1' then R[n]<7:0> else R[m]<7:0>;
    R[d]<15:8> = if APSR.GE<1> == '1' then R[n]<15:8> else R[m]<15:8>;
    R[d]<23:16> = if APSR.GE<2> == '1' then R[n]<23:16> else R[m]<23:16>;
    R[d]<31:24> = if APSR.GE<3> == '1' then R[n]<31:24> else R[m]<31:24>;
```

Restrictions

None.

Condition flags

These instructions do not change the flags.

Example 3-11 Examples

```
SADD16 R0, R1, R2     ; Set GE bits based on result.
SEL R0, R0, R3        ; Select bytes from R0 or R3, based on GE.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.13 SHADD16 and SHADD8

Signed Halving Add 16 and Signed Halving Add 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	SHADD16	Signed Halving Add 16.
	SHADD8	Signed Halving Add 8.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register.

The SHADD16 instruction:The SHADD8 instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halfword results in the destination register.

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the byte results in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-12 Examples

```
SHADD16 R1, R0     ; Adds halfwords in R0 to corresponding halfword of R1 and
                   ; writes halved result to corresponding halfword in R1.
SHADD8  R4, R0, R5 ; Adds bytes of R0 to corresponding byte in R5 and
                   ; writes halved result to corresponding byte in R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.14 SHASX and SHSAX

Signed Halving Add and Subtract with Exchange and Signed Halving Subtract and Add with Exchange.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	`SHASX`	Add and Subtract with Exchange and Halving.
	`SHSAX`	Subtract and Add with Exchange and Halving.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

The `SHASX` instruction:

1. Adds the top halfword of the first operand with the bottom halfword of the second operand.
2. Writes the halfword result of the addition to the top halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.
3. Subtracts the top halfword of the second operand from the bottom highword of the first operand.
4. Writes the halfword result of the division in the bottom halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.

The `SHSAX` instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Writes the halfword result of the addition to the bottom halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.
3. Adds the bottom halfword of the first operand with the top halfword of the second operand.
4. Writes the halfword result of the division in the top halfword of the destination register, shifted by one bit to the right causing a divide by two, or halving.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Example 3-13 Examples

```
SHASX    R7, R4, R2      ; Adds top halfword of R4 to bottom halfword of R2
                         ; and writes halved result to top halfword of R7.
                         ; Subtracts top halfword of R2 from bottom halfword of
                         ; R4 and writes halved result to bottom halfword of R7.
SHSAX   R0, R3, R5       ; Subtracts bottom halfword of R5 from top halfword
                         ; of R3 and writes halved result to top halfword of R0.
                         ; Adds top halfword of R5 to bottom halfword of R3 and
                         ; writes halved result to bottom halfword of R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.15 SHSUB16 and SHSUB8

Signed Halving Subtract 16 and Signed Halving Subtract 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	SHSUB16	Signed Halving Subtract 16.
	SHSUB8	Signed Halving Subtract 8.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register.

The SHSUB16 instruction:

1. Subtracts each halfword of the second operand from the corresponding halfwords of the first operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halved halfword results in the destination register.

The SHSUBB8 instruction:

1. Subtracts each byte of the second operand from the corresponding byte of the first operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the corresponding signed byte results in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-14 Examples

```
SHSUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword
                    ; of R1 and writes to corresponding halfword of R1.
SHSUB8  R4, R0, R5  ; Subtracts bytes of R0 from corresponding byte in R5,
                    ; and writes to corresponding byte in R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.16 SSUB16 and SSUB8

Signed Subtract 16 and Signed Subtract 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	`SSUB16`	Performs two 16-bit signed integer subtractions.
	`SSUB8`	Performs four 8-bit signed integer subtractions.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to change endianness of data.

The `SSUB16` instruction:The `SSUB8` instruction:

1. Subtracts each halfword from the second operand from the corresponding halfword of the first operand.
2. Writes the difference result of two signed halfwords in the corresponding halfword of the destination register.
1. Subtracts each byte of the second operand from the corresponding byte of the first operand.
2. Writes the difference result of four signed bytes in the corresponding byte of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results of the subtractions.

For `SSUB16`:

```
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]<15:0> = diff1<15:0>;
   R[d]<31:16> = diff2<15:0>;

   APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';

   APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

For `SSUB8`:

```
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> = diff1<7:0>;
   R[d]<15:8> = diff2<7:0>;
   R[d]<23:16> = diff3<7:0>;
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
   R[d]<31:24> = diff4<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';
```

Example 3-15 Examples

```
SSUB16 R1, R0       ; Subtracts halfwords in R0 from corresponding halfword of R1
                    ; and writes to corresponding halfword of R1.
SSUB8  R4, R0, R5   ; Subtracts bytes of R5 from corresponding byte in
                    ; R0, and writes to corresponding byte of R4.
```

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.17 TST and TEQ

Test bits and Test Equivalence.

Syntax

`TST{cond} Rn, Operand2`

`TEQ{cond} Rn, Operand2`

Where:

cond	Is an optional condition code.
Rn	Is the first operand register.
Operand2	Is a flexible second operand.

Operation

These instructions test the value in a register against *Operand2*. They update the condition flags based on the result, but do not write the result to a register.

The `TST` instruction performs a bitwise AND operation on the value in *Rn* and the value of *Operand2*. This is the same as the `ANDS` instruction, except that it discards the result.

To test whether a bit of *Rn* is 0 or 1, use the `TST` instruction with an *Operand2* constant that has that bit set to 1 and all other bits cleared to 0.

The `TEQ` instruction performs a bitwise Exclusive OR operation on the value in *Rn* and the value of *Operand2*. This is the same as the `EORS` instruction, except that it discards the result.

Use the `TEQ` instruction to test if two values are equal without affecting the V or C flags.

`TEQ` is also useful for testing the sign of a value. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions:

- Update the N and Z flags according to the result.
- Can update the C flag during the calculation of *Operand2*,
- Do not affect the V flag.

Example 3-16 Examples

```
TST     R0, #0x3F8  ; Perform bitwise AND of R0 value to 0x3F8,
                    ; APSR is updated but result is discarded
TEQEQ   R10, R9     ; Conditionally test if value in R10 is equal to
                    ; value in R9, APSR is updated but result is discarded.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.18 UADD16 and UADD8

Unsigned Add 16 and Unsigned Add 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	`UADD16`	Performs two 16-bit unsigned integer additions.
	`UADD8`	Performs four 8-bit unsigned integer additions.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to add 16- and 8-bit unsigned data.

The `UADD16` instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Writes the unsigned result in the corresponding halfwords of the destination register.

The `UADD8` instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Writes the unsigned result in the corresponding byte of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results of the additions.

For `UADD16`:

```
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]<15:0> = sum1<15:0>;
   R[d]<31:16> = sum2<15:0>;
   APSR.GE<1:0> = if sum1 >= 0x10000 then '11' else '00';
   APSR.GE<3:2> = if sum2 >= 0x10000 then '11' else '00';
```

For `UADD8`:

```
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]<7:0> = sum1<7:0>;
   R[d]<15:8> = sum2<7:0>;
   R[d]<23:16> = sum3<7:0>;
   R[d]<31:24> = sum4<7:0>;
   APSR.GE<0> = if sum1 >= 0x100 then '1' else '0';
   APSR.GE<1> = if sum2 >= 0x100 then '1' else '0';
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
   APSR.GE<2> = if sum3 >= 0x100 then '1' else '0';
   APSR.GE<3> = if sum4 >= 0x100 then '1' else '0';
```

Example 3-17 Examples

```
UADD16 R1, R0      ; Adds halfwords in R0 to corresponding halfword of R1,
                   ; writes to corresponding halfword of R1.
UADD8  R4, R0, R5  ; Adds bytes of R0 to corresponding byte in R5 and writes
                   ; to corresponding byte in R4.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.19 UASX and USAX

Unsigned Add and Subtract with Exchange and Unsigned Subtract and Add with Exchange.

Syntax

`op{cond} {Rd,} Rn, Rm`

Where:

op	Is one of:	
	`UASX`	Add and Subtract with Exchange.
	`USAX`	Subtract and Add with Exchange.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

The `UASX` instruction:

1. Subtracts the top halfword of the second operand from the bottom halfword of the first operand.
2. Writes the unsigned result from the subtraction to the bottom halfword of the destination register.
3. Adds the top halfword of the first operand with the bottom halfword of the second operand.
4. Writes the unsigned result of the addition to the top halfword of the destination register.

The `USAX` instruction:

1. Adds the bottom halfword of the first operand with the top halfword of the second operand.
2. Writes the unsigned result of the addition to the bottom halfword of the destination register.
3. Subtracts the bottom halfword of the second operand from the top halfword of the first operand.
4. Writes the unsigned result from the subtraction to the top halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results.

For `UASX`:

```
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]<15:0> = diff<15:0>;
   R[d]<31:16> = sum<15:0>;
   APSR.GE<1:0> = if diff >= 0 then '11' else '00';
   APSR.GE<3:2> = if sum >= 0x10000 then '11' else '00';
```

For `USAX`:

```
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]<15:0> = sum<15:0>;
   R[d]<31:16> = diff<15:0>;
   APSR.GE<1:0> = if sum >= 0x10000 then '11' else '00';
   APSR.GE<3:2> = if diff >= 0 then '11' else '00';
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-18 Examples

```
UASX    R0, R4, R5     ; Adds top halfword of R4 to bottom halfword of R5 and
                       ; writes to top halfword of R0.
                       ; Subtracts bottom halfword of R5 from top halfword of R0
                       ; and writes to bottom halfword of R0.
USAX    R7, R3, R2     ; Subtracts top halfword of R2 from bottom halfword of R3
                       ; and writes to bottom halfword of R7.
                       ; Adds top halfword of R3 to bottom halfword of R2 and
                       ; writes to top halfword of R7.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.20 UHADD16 and UHADD8

Unsigned Halving Add 16 and Unsigned Halving Add 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op
: Is one of:

`UHADD16`	Unsigned Halving Add 16.
`UHADD8`	Unsigned Halving Add 8.

cond
: Is an optional condition code.

Rd
: Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn
: Is the register holding the first operand.

Rm
: Is the register holding the second operand.

Operation

Use these instructions to add 16- and 8-bit data and then to halve the result before writing the result to the destination register.

The `UHADD16` instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Shuffles the halfword result by one bit to the right, halving the data.
3. Writes the unsigned results to the corresponding halfword in the destination register.

The `UHADD8` instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Shuffles the byte result by one bit to the right, halving the data.
3. Writes the unsigned results in the corresponding byte in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-19 Examples

```
UHADD16 R7, R3      ; Adds halfwords in R7 to corresponding halfword of R3
                    ; and writes halved result to corresponding halfword in R7.
UHADD8  R4, R0, R5  ; Adds bytes of R0 to corresponding byte in R5 and writes
                    ; halved result to corresponding byte in R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.21 UHASX and UHSAX

Unsigned Halving Add and Subtract with Exchange and Unsigned Halving Subtract and Add with Exchange.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	`UHASX`	Unsigned Halving Add and Subtract with Exchange.
	`UHSAX`	Unsigned Halving Subtract and Add with Exchange.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

The `UHASX` instruction:

1. Adds the top halfword of the first operand with the bottom halfword of the second operand.
2. Shifts the result by one bit to the right causing a divide by two, or halving.
3. Writes the halfword result of the addition to the top halfword of the destination register.
4. Subtracts the top halfword of the second operand from the bottom halfword of the first operand.
5. Shifts the result by one bit to the right causing a divide by two, or halving.
6. Writes the halfword result of the subtraction in the bottom halfword of the destination register.

The `UHSAX` instruction:

1. Subtracts the bottom halfword of the second operand from the top halfword of the first operand.
2. Shifts the result by one bit to the right causing a divide by two, or halving.
3. Writes the halfword result of the subtraction in the top halfword of the destination register.
4. Adds the bottom halfword of the first operand with the top halfword of the second operand.
5. Shifts the result by one bit to the right causing a divide by two, or halving.
6. Writes the halfword result of the addition to the bottom halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Example 3-20 Examples

```
UHASX    R7, R4, R2     ; Adds top halfword of R4 with bottom halfword of R2
                        ; and writes halved result to top halfword of R7.
                        ; Subtracts top halfword of R2 from bottom halfword of
                        ; R7 and writes halved result to bottom halfword of R7.
UHSAX    R0, R3, R5     ; Subtracts bottom halfword of R5 from top halfword of
                        ; R3 and writes halved result to top halfword of R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        ; Adds top halfword of R5 to bottom halfword of R3 and
                        ; writes halved result to bottom halfword of R0.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.22 UHSUB16 and UHSUB8

Unsigned Halving Subtract 16 and Unsigned Halving Subtract 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op
Is one of:

`UHSUB16` Performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

`UHSUB8` Performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

cond
Is an optional condition code.

Rd
Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn
Is the first operand register.

Rm
Is the second operand register.

Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register.

The `UHSUB16` instruction:

1. Subtracts each halfword of the second operand from the corresponding halfword of the first operand.
2. Shuffles each halfword result to the right by one bit, halving the data.
3. Writes each unsigned halfword result to the corresponding halfwords in the destination register.

The `UHSUB8` instruction:

1. Subtracts each byte of second operand from the corresponding byte of the first operand.
2. Shuffles each byte result by one bit to the right, halving the data.
3. Writes the unsigned byte results to the corresponding byte of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-21 Examples

```
UHSUB16  R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of
                     ; R1 and writes halved result to corresponding halfword in
                     ; R1.
UHSUB8  R4, R0, R5   ; Subtracts bytes of R5 from corresponding byte in R0 and
                     ; writes halved result to corresponding byte in R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.23 USAD8

Unsigned Sum of Absolute Differences.

Syntax

`USAD8{cond} {Rd,} Rn, Rm`

Where:

cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn	Is the first operand register.
Rm	Is the second operand register.

Operation

The `USAD8` instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Adds the absolute values of the differences together.
3. Writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-22 Examples

```
USAD8 R1, R4, R0      ; Subtracts each byte in R0 from corresponding byte of R4
                      ; adds the differences and writes to R1.
USAD8  R0, R5         ; Subtracts bytes of R5 from corresponding byte in R0,
                      ; adds the differences and writes to R0.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.24 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

Syntax

`USADA8{cond} Rd, Rn, Rm, Ra`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
Rn	Is the first operand register.
Rm	Is the second operand register.
Ra	Is the register that contains the accumulation value.

Operation

The `USADA8` instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Adds the unsigned absolute differences together.
3. Adds the accumulation value to the sum of the absolute differences.
4. Writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-23 Examples

```
USADA8 R1, R0, R6      ; Subtracts bytes in R0 from corresponding halfword of R1
                       ; adds differences, adds value of R6, writes to R1.
USADA8  R4, R0, R5, R2 ; Subtracts bytes of R5 from corresponding byte in R0
                       ; adds differences, adds value of R2 writes to R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.13.25 USUB16 and USUB8

Unsigned Subtract 16 and Unsigned Subtract 8.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	`USUB16`	Unsigned Subtract 16.
	`USUB8`	Unsigned Subtract 8.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn	Is the first operand register.	
Rm	Is the second operand register.	

Operation

Use these instructions to subtract 16-bit and 8-bit data before writing the result to the destination register.

The `USUB16` instruction:

1. Subtracts each halfword from the second operand register from the corresponding halfword of the first operand register.
2. Writes the unsigned result in the corresponding halfwords of the destination register.

The `USUB8` instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Writes the unsigned byte result in the corresponding byte of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results of the subtractions.

For `USUB16`:

```
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]<15:0> = diff1<15:0>;
   R[d]<31:16> = diff2<15:0>;
   APSR.GE<1:0> = if diff1 >= 0 then '11' else '00';
   APSR.GE<3:2> = if diff2 >= 0 then '11' else '00';
```

For `USUB8`:

```
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]<7:0> = diff1<7:0>;
   R[d]<15:8> = diff2<7:0>;
   R[d]<23:16> = diff3<7:0>;
   R[d]<31:24> = diff4<7:0>;
   APSR.GE<0> = if diff1 >= 0 then '1' else '0';
   APSR.GE<1> = if diff2 >= 0 then '1' else '0';
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
   APSR.GE<2> = if diff3 >= 0 then '1' else '0';
   APSR.GE<3> = if diff4 >= 0 then '1' else '0';
```

Example 3-24 Examples

```
USUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of R1
                   ; and writes to corresponding halfword in R1.
USUB8  R4, R0, R5  ; Subtracts bytes of R5 from corresponding byte in R0 and
                   ; writes to the corresponding byte in R4.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14 Coprocessor instructions

Reference material for the Cortex-M55 processor coprocessor instruction set.

3.14.1 List of coprocessor instructions

An alphabetically ordered list of the coprocessor instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-7 Coprocessor instructions

Mnemonic	Brief description	See
CDP, CDP2	Coprocessor data processing	*3.14.3 CDP and CDP2* on page 3-164
MCR, MCR2	Move to Coprocessor from Register	*3.14.4 MCR and MCR2* on page 3-165
MCRR, MCRR2	Move to Coprocessor from two Registers	*3.14.5 MCRR and MCRR2* on page 3-166
MRC, MRC2	Move to Register from Coprocessor	*3.14.6 MRC and MRC2* on page 3-167
MRRC, MRRC2	Move to two Registers from Coprocessor	*3.14.7 MRRC and MRRC2* on page 3-168

3.14.2 Coprocessor intrinsics

The following table shows intrinsics for coprocessor data-processing instructions.

Intrinsics	Equivalent Instruction
void __arm_cdp(coproc, opc1, CRd, CRn, CRm, opc2)	CDP coproc, #opc1, CRd, CRn, CRm, #opc2
void __arm_cdp2(coproc, opc1, CRd, CRn, CRm, opc2)	CDP2 coproc, #opc1, CRd, CRn, CRm, #opc2

The following table shows intrinsics that map to coprocessor to core register transfer instructions.

Intrinsics	Equivalent Instruction
void __arm_mcr(coproc, opc1, uint32_t value, CRn, CRm, opc2)	MCR coproc, #opc1, Rt, CRn, CRm, #opc2
void __arm_mcr2(coproc, opc1, uint32_t value, CRn, CRm, opc2)	MCR2 coproc, #opc1, Rt, CRn, CRm, #opc2
uint32_t __arm_mrc(coproc, opc1, CRn, CRm, opc2)	MRC coproc, #opc1, Rt, CRn, CRm, #opc2
uint32_t __arm_mrc2(coproc, opc1, CRn, CRm, opc2)	MRC2 coproc, #opc1, Rt, CRn, CRm, #opc2
void __arm_mcrr(coproc, opc1, uint64_t value, CRm)	MCRR coproc, #opc1, Rt, Rt2, CRm
void __arm_mcrr2(coproc, opc1, uint64_t value, CRm)	MCRR2 coproc, #opc1, Rt, Rt2, CRm
uint64_t __arm_mrrc(coproc, opc1, CRm)	MRRC coproc, #opc1, Rt, Rt2, CRm
uint64_t __arm_mrrc2(coproc, opc1, CRm)	MRRC2 coproc, #opc1, Rt, Rt2, CRm

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14.3 CDP and CDP2

Coprocessor Data Processing tells a coprocessor to perform an operation.

Syntax

```
CDP{cond} coproc, #opc1, CRd, CRn, CRm{, #opc2}
```

```
CDP2{cond} coproc, #opc1, CRd, CRn, CRm{, #opc2}
```

Where:

cond is an optional condition code.

coproc is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be in the range 0-7.

opc1 is a 4-bit coprocessor-specific opcode.

opc2 is an optional 3-bit coprocessor-specific opcode.

CRd, CRn, CRm are coprocessor registers.

Operation

The operation of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14.4 MCR and MCR2

Move to Coprocessor from Register. Depending on the coprocessor, you might be able to specify various additional operations.

Syntax

```
MCR{cond} coproc, #opc1, Rt, CRn, CRm{, #opc2}
```

```
MCR2{cond} coproc, #opc1, Rt, CRn, CRm{, #opc2}
```

where:

cond is an optional condition code.

coproc is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be In the range 0-7.

opc1 is a 3-bit coprocessor-specific opcode.

opc2 is an optional 3-bit coprocessor-specific opcode.

Rt is an Arm source register. `Rt` must not be PC.

CRn, CRm are coprocessor registers.

Operation

The operation of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14.5 MCRR and MCRR2

Move to Coprocessor from two Registers. Depending on the coprocessor, you might be able to specify various additional operations.

Syntax

```
MCRR{cond} coproc, #opc1, Rt, Rt2, CRm
```

```
MCRR2{cond} coproc, #opc1, Rt, Rt2, CRm
```

Where:

cond is an optional condition code.

coproc is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be In the range 0-7.

opc1 is a 3-bit coprocessor-specific opcode.

Rt, Rt2 are Arm source registers. Rt and Rt2 must not be PC.

CRm are coprocessor registers.

Operation

The operation of these instructions depends on the coprocessor. See the coprocessor documentation for details.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14.6 MRC and MRC2

Move to Register from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

Syntax

`MRC{cond} coproc, #opc1, Rt, CRn, CRm{, #opc2}`

`MRC2{cond} coproc, #opc1, Rt, CRn, CRm{, #opc2}`

where:

cond is an optional condition code.

coproc is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be in the range 0-7.

opc1 is a 3-bit coprocessor-specific opcode.

opc2 is an optional 3-bit coprocessor-specific opcode.

Rt is the Arm destination register. `Rt` must not be PC.

Rt can be `APSR_nzcv`. This means that the coprocessor executes an instruction that changes the value of the condition flags in the APSR.

CRn, CRm are coprocessor registers.

Operation

The operation of these instructions depends on the coprocessor. See the coprocessor documentation for details.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.14.7 MRRC and MRRC2

Move to two Registers from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

Syntax

```
MRRC{cond} coproc, #opc1, Rt, Rt2, CRm
```

```
MRRC2{cond} coproc, #opc1, Rt, Rt2, CRm
```

Where:

cond is an optional condition code.

coproc is the name of the coprocessor the instruction is for. The standard name is p*n*, where *n* is an integer whose value must be in the range 0-7.

opc1 is a 3-bit coprocessor-specific opcode.

Rt, Rt2 are Arm destination registers. `Rt` and `Rt2` must not be PC.

CRm is a coprocessor register.

Operation

The operation of these instructions depends on the coprocessor. See the coprocessor documentation for details.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15 Multiply and divide instructions

Reference material for the Cortex-M55 processor multiply and divide instruction set.

3.15.1 List of multiply and divide instructions

An alphabetically ordered list of the multiply and divide instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-8 Multiply and divide instructions

Mnemonic	Brief description	See
MLA	Multiply with Accumulate, 32-bit result	*3.15.2 MUL, MLA, and MLS* on page 3-171
MLS	Multiply and Subtract, 32-bit result	*3.15.2 MUL, MLA, and MLS* on page 3-171
MUL	Multiply, 32-bit result	*3.15.2 MUL, MLA, and MLS* on page 3-171
SDIV	Signed Divide	*3.15.3 SDIV and UDIV* on page 3-172
SMLA[B,T]	Signed Multiply Accumulate (halfwords)	*3.15.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-173
SMLAD, SMLADX	Signed Multiply Accumulate Dual	*3.15.5 SMLAD and SMLADX* on page 3-175
SMLAL	Signed Multiply with Accumulate (32 × 32 + 64), 64-bit result	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
SMLAL[B,T]	Signed Multiply Accumulate Long (halfwords)	*3.15.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-177
SMLALD, SMLALDX	Signed Multiply Accumulate Long Dual	*3.15.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-177
SMLAW[B|T]	Signed Multiply Accumulate (word by halfword)	*3.15.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-173
SMLSD	Signed Multiply Subtract Dual	*3.15.7 SMLSD and SMLSLD* on page 3-179
SMLSLD	Signed Multiply Subtract Long Dual	*3.15.7 SMLSD and SMLSLD* on page 3-179
SMMLA	Signed Most Significant Word Multiply Accumulate	*3.15.8 SMMLA and SMMLS* on page 3-181
SMMLS, SMMLSR	Signed Most Significant Word Multiply Subtract	*3.15.8 SMMLA and SMMLS* on page 3-181
SMMUL, SMMULR	Signed Most Significant Word Multiply	*3.15.9 SMMUL* on page 3-183
SMUAD, SMUADX	Signed Dual Multiply Add	*3.15.10 SMUAD and SMUSD* on page 3-184
SMUL[B,T]	Signed Multiply (word by halfword)	*3.15.11 SMUL and SMULW* on page 3-186
SMULL	Signed Multiply (32 × 32), 64-bit result	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
SMULWB, SMULWT	Signed Multiply (word by halfword)	*3.15.11 SMUL and SMULW* on page 3-186
SMUSDX ,SMUSD	Signed Dual Multiply Subtract	*3.15.10 SMUAD and SMUSD* on page 3-184
UDIV	Unsigned Divide	*3.15.3 SDIV and UDIV* on page 3-172
UMAAL	Unsigned Multiply Accumulate Accumulate Long (32 × 32 + 32 + 32), 64-bit result	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-8 Multiply and divide instructions (continued)

Mnemonic	Brief description	See
UMLAL	Unsigned Multiply with Accumulate (32 × 32 + 64), 64-bit result	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188
UMULL	Unsigned Multiply (32 × 32), 64-bit result	*3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL* on page 3-188

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.2 MUL, MLA, and MLS

Multiply, Multiply with Accumulate, and Multiply with Subtract, using 32-bit operands, and producing a 32-bit result.

Syntax

`MUL{S}{cond} {Rd,} Rn, Rm ; Multiply`

`MLA{cond} Rd, Rn, Rm, Ra ; Multiply with accumulate`

`MLS{cond} Rd, Rn, Rm, Ra ; Multiply with subtract`

Where:

cond	Is an optional condition code.
S	Is an optional suffix. If `S` is specified, the condition code flags are updated on the result of the operation.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn, Rm	Are registers holding the values to be multiplied.
Ra	Is a register holding the value to be added or subtracted from.

Operation

The `MUL` instruction multiplies the values from *Rn* and *Rm*, and places the least significant 32 bits of the result in *Rd*.

The `MLA` instruction multiplies the values from *Rn* and *Rm*, adds the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

The `MLS` instruction multiplies the values from *Rn* and *Rm*, subtracts the product from the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

The results of these instructions do not depend on whether the operands are signed or unsigned.

Restrictions

In these instructions, do not use SP and do not use PC.

If you use the `S` suffix with the `MUL` instruction:

- *Rd*, *Rn*, and *Rm* must all be in the range `R0-R7`.
- *Rd* must be the same as *Rm*.
- You must not use the *cond* suffix.

Condition flags

The MLA instruction and MULS instructions:

- Only MULS instruction updates the N and Z flags according to the result.
- No other MUL, MLA, or MLS instruction affects the condition flags.

Example 3-25 Examples

```
MUL     R10, R2, R5     ; Multiply, R10 = R2 × R5
MLA     R10, R2, R1, R5 ; Multiply with accumulate, R10 = (R2 × R1) + R5
MULS    R0, R2, R2      ; Multiply with flag update, R0 = R2 × R2
MULLT   R2, R3, R2      ; Conditionally multiply, R2 = R3 × R2
MLS     R4, R5, R6, R7  ; Multiply with subtract, R4 = R7 - (R5 × R6)
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.3 SDIV and UDIV

Signed Divide and Unsigned Divide.

Syntax

`SDIV{cond} {Rd,} Rn, Rm`

`UDIV{cond} {Rd,} Rn, Rm`

Where:

cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn	Is the register holding the value to be divided.
Rm	Is a register holding the divisor.

Operation

The `SDIV` instruction performs a signed integer division of the value in *Rn* by the value in *Rm*.

The `UDIV` instruction performs an unsigned integer division of the value in *Rn* by the value in *Rm*.

For both instructions, if the value in *Rn* is not divisible by the value in *Rm*, the result is rounded towards zero.

For the Cortex-M55 processor, the integer divide operation latency is in the range of 2-11 cycles.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Example 3-26 Examples

```
SDIV  R0, R2, R4  ; Signed divide, R0 = R2/R4
UDIV  R8, R8, R1  ; Unsigned divide, R8 = R8/R1
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT

Signed Multiply Accumulate (halfwords).

Syntax

`op{cond} Rd, Rn, Rm, Ra`

Where:

op	Is one of:	
	`SMLAWB`	Signed Multiply Accumulate (word by halfword) The bottom halfword, bits [15:0], of *Rm* is used.
	`SMLAWT`	Signed Multiply Accumulate (word by halfword) The top halfword, bits [31:16] of *Rm* is used.
	`SMLABB`, `SMLABT`	Signed Multiply Accumulate Long (halfwords) The bottom halfword, bits [15:0], of *Rm* is used.
	`SMLATB`, `SMLATT`	Signed Multiply Accumulate Long (halfwords) The top halfword, bits [31:16] of *Rm* is used.
cond	Is an optional condition code.	
Rd	Is the destination register.	
Rn, Rm	Are registers holding the values to be multiplied.	
Ra	Is a register holding the value to be added or subtracted from.	

Operation

The `SMLABB`, `SMLABT`, `SMLATB`, `SMLATT` instructions:

- Multiply the specified signed halfword, top or bottom, values from *Rn* and *Rm*.
- Add the value in *Ra* to the resulting 32-bit product.
- Write the result of the multiplication and addition in *Rd*.

The non-specified halfwords of the source registers are ignored.

The `SMLAWB` and `SMLAWT` instructions:

- Multiply the 32-bit signed values in *Rn* with:
  — The top signed halfword of *Rm*, `T` instruction suffix.
  — The bottom signed halfword of *Rm*, `B` instruction suffix.
- Add the 32-bit signed value in *Ra* to the top 32 bits of the 48-bit product
- Write the result of the multiplication and addition in *Rd*.

The bottom 16 bits of the 48-bit product are ignored.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

If overflow occurs during the addition of the accumulate value, the `SMLAWB`, `SMLAWT`, instruction sets the Q flag in the APSR. No overflow can occur during the multiplication.

Restrictions

In these instructions, do not use SP and do not use PC.

Condition flags

If an overflow is detected, the Q flag is set.

Example 3-27 Examples

```
SMLABB  R5, R6, R4, R1  ; Multiplies bottom halfwords of R6 and R4, adds
                        ; R1 and writes to R5.
SMLATB  R5, R6, R4, R1  ; Multiplies top halfword of R6 with bottom halfword
                        ; of R4, adds R1 and writes to R5.
SMLATT  R5, R6, R4, R1  ; Multiplies top halfwords of R6 and R4, adds
                        ; R1 and writes the sum to R5.
SMLABT  R5, R6, R4, R1  ; Multiplies bottom halfword of R6 with top halfword
                        ; of R4, adds R1 and writes to R5.
SMLABT  R4, R3, R2      ; Multiplies bottom halfword of R4 with top halfword of
                        ; R3, adds R2 and writes to R4.
SMLAWB  R10, R2, R5, R3 ; Multiplies R2 with bottom halfword of R5, adds
                        ; R3 to the result and writes top 32-bits to R10.
SMLAWT  R10, R2, R1, R5 ; Multiplies R2 with top halfword of R1, adds R5
                        ; and writes top 32-bits to R10.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.5 SMLAD and SMLADX

Signed Multiply Accumulate Long Dual, Signed Multiply Accumulate Long Dual exchange.

Syntax

`op{X}{cond} Rd, Rn, Rm, Ra`

Where:

op
Is one of:

`SMLAD` Signed Multiply Accumulate Long Dual.

`SMLADX` Signed Multiply Accumulate Long Dual exchange.

`X` specifies which halfword of the source register *Rn* is used as the multiply operand.

If `X` is omitted, the multiplications are bottom × bottom and top × top.

If `X` is present, the multiplications are bottom × top and top × bottom.

cond
Is an optional condition code.

Rd
Is the destination register.

Rn
Is the first operand register holding the values to be multiplied.

Rm
Is the second operand register.

Ra
Is the accumulate value.

Operation

The `SMLAD` and `SMLADX` instructions regard the two operands as four halfword 16-bit values.

The `SMLAD` instruction:

1. Multiplies the top signed halfword value in *Rn* with the top signed halfword of *Rm* and the bottom signed halfword value in *Rn* with the bottom signed halfword of *Rm*.
2. Adds both multiplication results to the signed 32-bit value in *Ra*.
3. Writes the 32-bit signed result of the multiplication and addition to *Rd*.

The `SMLADX` instruction:

1. Multiplies the top signed halfword value in *Rn* with the bottom signed halfword of *Rm* and the bottom signed halfword value in *Rn* with the top signed halfword of *Rm*.
2. Adds both multiplication results to the signed 32-bit value in *Ra*.
3. Writes the 32-bit signed result of the multiplication and addition to *Rd*.

Restrictions

Do not use SP and do not use PC.

Condition flags

Sets the Q flag if the accumulate operation overflows.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-28 Examples

```
SMLAD   R10, R2, R1, R5 ; Multiplies two halfword values in R2 with
                        ; corresponding halfwords in R1, adds R5 and writes to
                        ; R10.
SMLALDX R0, R2, R4, R6  ; Multiplies top halfword of R2 with bottom halfword
                        ; of R4, multiplies bottom halfword of R2 with top
                        ; halfword of R4, adds R6 and writes to R0.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT

Signed Multiply Accumulate Long Dual and Signed Multiply Accumulate Long (halfwords).

Syntax

op{cond} RdLo, RdHi, Rn, Rm

Where:

op

Is one of:

SMLALBB, SMLALBT	Signed Multiply Accumulate Long (halfwords, B and T). B and T specify which halfword of the source registers *Rn* and *Rm* are used as the first and second multiply operand: The bottom halfword, bits [15:0], of *Rn* is used. SMLALBB: the bottom halfword, bits [15:0], of *Rm* is used. SMLALBT: the top halfword, bits [31:16], of *Rm* is used.
SMLALTB, SMLALTT	Signed Multiply Accumulate Long (halfwords, B and T). The top halfword, bits [31:16], of *Rn* is used. SMLALTB: the bottom halfword, bits [15:0], of *Rm* is used. SMLALTT: the top halfword, bits [31:16], of *Rm* is used.
SMLALD	Signed Multiply Accumulate Long Dual. The multiplications are bottom × bottom and top × top.
SMLALDX	Signed Multiply Accumulate Long Dual reversed. The multiplications are bottom × top and top × bottom.

cond

Is an optional condition code.

RdHi, RdLo

Are the destination registers. *RdLo* is the lower 32 bits and *RdHi* is the upper 32 bits of the 64-bit integer. The accumulating value for the lower and upper 32 bits are held in the *RdLo* and *RdHi* registers respectively.

Rn, Rm

Are registers holding the first and second operands.

Operation

- Multiplies the two's complement signed word values from *Rn* and *Rm*.
- Adds the 64-bit value in *RdLo* and *RdHi* to the resulting 64-bit product.
- Writes the 64-bit result of the multiplication and addition in *RdLo* and *RdHi*.

The SMLALBB, SMLALBT, SMLALTB and SMLALTT instructions:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- Multiplies the specified signed halfword, Top or Bottom, values from *Rn* and *Rm*.
- Adds the resulting sign-extended 32-bit product to the 64-bit value in *RdLo* and *RdHi*.
- Writes the 64-bit result of the multiplication and addition in *RdLo* and *RdHi*.

The non-specified halfwords of the source registers are ignored.

The `SMLALD` and `SMLALDX` instructions interpret the values from *Rn* and *Rm* as four halfword two's complement signed 16-bit integers. These instructions:

- `SMLALD` multiplies the top signed halfword value of *Rn* with the top signed halfword of *Rm* and the bottom signed halfword values of *Rn* with the bottom signed halfword of *Rm*.
- `SMLALDX` multiplies the top signed halfword value of *Rn* with the bottom signed halfword of *Rm* and the bottom signed halfword values of *Rn* with the top signed halfword of *Rm*.
- Add the two multiplication results to the signed 64-bit value in *RdLo* and *RdHi* to create the resulting 64-bit product.
- Write the 64-bit product in *RdLo* and *RdHi*.

Restrictions

In these instructions:

- Do not use SP and do not use PC.
- *RdHi* and *RdLo* must be different registers.

Condition flags

These instructions do not affect the condition code flags.

Example 3-29 Examples

```
SMLALBT     R2, R1, R6, R7   ; Multiplies bottom halfword of R6 with top
                             ; halfword of R7, sign extends to 32-bit, adds
                             ; R1:R2 and writes to R1:R2.
SMLALTB     R2, R1, R6, R7   ; Multiplies top halfword of R6 with bottom
                             ; halfword of R7,sign extends to 32-bit, adds R1:R2
                             ; and writes to R1:R2.
SMLALD      R6, R8, R5, R1   ; Multiplies top halfwords in R5 and R1 and bottom
                             ; halfwords of R5 and R1, adds R8:R6 and writes to
                             ; R8:R6.
SMLALDX     R6, R8, R5, R1   ; Multiplies top halfword in R5 with bottom
                             ; halfword of R1, and bottom halfword of R5 with
                             ; top halfword of R1, adds R8:R6 and writes to
                             ; R8:R6.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.7 SMLSD and SMLSLD

Signed Multiply Subtract Dual and Signed Multiply Subtract Long Dual.

Syntax

`op{X}{cond} Rd, Rn, Rm, Ra ; SMLSD`

`op{X}{cond} RdLo, RdHi, Rn, Rm ; SMLSLD`

Where:

op	Is one of: `SMLSD` Signed Multiply Subtract Dual. `SMLSDX` Signed Multiply Subtract Dual reversed. `SMLSLD` Signed Multiply Subtract Long Dual. `SMLSLDX` Signed Multiply Subtract Long Dual reversed. If `X` is present, the multiplications are bottom × top and top × bottom. If the `X` is omitted, the multiplications are bottom × bottom and top × top.
cond	Is an optional condition code.
Rd	Is the destination register.
Rn, Rm	Are registers holding the first and second operands.
Ra	Is the register holding the accumulate value.
RdLo	Supplies the lower 32 bits of the accumulate value, and is the destination register for the lower 32 bits of the result.
RdHi	Supplies the upper 32 bits of the accumulate value, and is the destination register for the upper 32 bits of the result.

Operation

The `SMLSD` instruction interprets the values from the first and second operands as four signed halfwords. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit halfword multiplications.
- Subtracts the result of the upper halfword multiplication from the result of the lower halfword multiplication.
- Adds the signed accumulate value to the result of the subtraction.
- Writes the result of the addition to the destination register.

The `SMLSLD` instruction interprets the values from *Rn* and *Rm* as four signed halfwords. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit halfword multiplications.
- Subtracts the result of the upper halfword multiplication from the result of the lower halfword multiplication.
- Adds the 64-bit value in *RdHi* and *RdLo* to the result of the subtraction.
- Writes the 64-bit result of the addition to the *RdHi* and *RdLo*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

In these instructions:

- Do not use SP and do not use PC.

Condition flags

The `SMLSD{X}` instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

For the T32 instruction set, these instructions do not affect the condition code flags.

Example 3-30 Examples

```
SMLSD       R0, R4, R5, R6   ; Multiplies bottom halfword of R4 with bottom
                             ; halfword of R5, multiplies top halfword of R4
                             ; with top halfword of R5, subtracts second from
                             ; first, adds R6, writes to R0.
SMLSDX      R1, R3, R2, R0   ; Multiplies bottom halfword of R3 with top
                             ; halfword of R2, multiplies top halfword of R3
                             ; with bottom halfword of R2, subtracts second from
                             ; first, adds R0, writes to R1.
SMLSLD      R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with bottom
                             ; halfword of R2, multiplies top halfword of R6
                             ; with top halfword of R2, subtracts second from
                             ; first, adds R6:R3, writes to R6:R3.
SMLSLDX     R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with top
                             ; halfword of R2, multiplies top halfword of R6
                             ; with bottom halfword of R2, subtracts second from
                             ; first, adds R6:R3, writes to R6:R3.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.8 SMMLA and SMMLS

Signed Most Significant Word Multiply Accumulate and Signed Most Significant Word Multiply Subtract.

Syntax

op{R}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

Where:

op	Is one of: `SMMLA` Signed Most Significant Word Multiply Accumulate. `SMMLS` Signed Most Significant Word Multiply Subtract.
R	If `R` is present, the result is rounded instead of being truncated. In this case the constant `0x80000000` is added to the product before the top halfword is extracted.
cond	Is an optional condition code.
Rd	Is the destination register.
Rn, *Rm*	Are registers holding the first and second multiply operands.
Ra	Is the register holding the accumulate value.

Operation

The `SMMLA` instruction interprets the values from *Rn* and *Rm* as signed 32-bit words.

The `SMMLA` instruction:

- Multiplies the values in *Rn* and *Rm*.
- Optionally rounds the result by adding `0x80000000`.
- Extracts the most significant 32 bits of the result.
- Adds the value of *Ra* to the signed extracted value.
- Writes the result of the addition in *Rd*.

The `SMMLS` instruction interprets the values from *Rn* and *Rm* as signed 32-bit words.

The `SMMLS` instruction:

- Multiplies the values in *Rn* and *Rm*.
- Optionally rounds the result by adding `0x80000000`.
- Extracts the most significant 32 bits of the result.
- Subtracts the extracted value of the result from the value in *Ra*.
- Writes the result of the subtraction in *Rd*.

Restrictions

In these instructions:

- Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Example 3-31 Examples

```
SMMLA     R0, R4, R5, R6   ; Multiplies R4 and R5, extracts top 32 bits, adds
                           ; R6, truncates and writes to R0.
SMMLAR    R6, R2, R1, R4   ; Multiplies R2 and R1, extracts top 32 bits, adds
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                           ; R4, rounds and writes to R6.
SMMLSR     R3, R6, R2, R7  ; Multiplies R6 and R2, extracts top 32 bits,
                           ; subtracts R7, rounds and writes to R3.
SMMLS      R4, R5, R3, R8  ; Multiplies R5 and R3, extracts top 32 bits,
                           ; subtracts R8, truncates and writes to R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.9 SMMUL

Signed Most Significant Word Multiply.

Syntax

`op{R}{cond} Rd, Rn, Rm`

Where:

op	Is one of: `SMMUL` Signed Most Significant Word Multiply.
`R`	If `R` is present, the result is rounded instead of being truncated. In this case the constant `0x80000000` is added to the product before the top halfword is extracted.
cond	Is an optional condition code.
Rd	Is the destination register.
Rn, Rm	Are registers holding the first and second operands.

Operation

The `SMMUL` instruction interprets the values from *Rn* and *Rm* as two's complement 32-bit signed integers. The `SMMUL` instruction:

- Multiplies the values from *Rn* and *Rm*.
- Optionally rounds the result, otherwise truncates the result.
- Writes the most significant signed 32 bits of the result in *Rd*.

Restrictions

In this instruction:

- Do not use SP and do not use PC.

Condition flags

This instruction does not affect the condition code flags.

Example 3-32 Examples

```
SMMUL       R0, R4, R5        ; Multiplies R4 and R5, truncates top 32 bits
                              ; and writes to R0.
SMMULR      R6, R2            ; Multiplies R6 and R2, rounds the top 32 bits
                              ; and writes to R6.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.10 SMUAD and SMUSD

Signed Dual Multiply Add and Signed Dual Multiply Subtract.

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*

Where:

op	Is one of: `SMUAD` Signed Dual Multiply Add. `SMUADX` Signed Dual Multiply Add reversed. `SMUSD` Signed Dual Multiply Subtract. `SMUSDX` Signed Dual Multiply Subtract reversed.
X	If `X` is present, the multiplications are bottom × top and top × bottom. If the `X` is omitted, the multiplications are bottom × bottom and top × top.
cond	Is an optional condition code.
Rd	Is the destination register.
Rn, *Rm*	Are registers holding the first and the second operands.

Operation

The `SMUAD` instruction interprets the values from the first and second operands as two signed halfwords in each operand. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit multiplications.
- Adds the two multiplication results together.
- Writes the result of the addition to the destination register.

The `SMUSD` instruction interprets the values from the first and second operands as two's complement signed integers. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 × 16-bit multiplications.
- Subtracts the result of the top halfword multiplication from the result of the bottom halfword multiplication.
- Writes the result of the subtraction to the destination register.

Restrictions

In these instructions:

- Do not use SP and do not use PC.

Condition flags

`SMUAD`, `SMUADX` set the Q flag if the addition overflows. The multiplications cannot overflow.

Example 3-33 Examples

```
SMUAD    R0, R4, R5      ; Multiplies bottom halfword of R4 with the bottom
                         ; halfword of R5, adds multiplication of top halfword
                         ; of R4 with top halfword of R5, writes to R0.
SMUADX   R3, R7, R4      ; Multiplies bottom halfword of R7 with top halfword
                         ; of R4, adds multiplication of top halfword of R7
                         ; with bottom halfword of R4, writes to R3.
SMUSD    R3, R6, R2      ; Multiplies bottom halfword of R4 with bottom halfword
                         ; of R6, subtracts multiplication of top halfword of R6
                         ; with top halfword of R3, writes to R3.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
SMUSDX   R4, R5, R3      ; Multiplies bottom halfword of R5 with top halfword of
                         ; R3, subtracts multiplication of top halfword of R5
                         ; with bottom halfword of R3, writes to R4.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.11 SMUL and SMULW

Signed Multiply (halfwords) and Signed Multiply (word by halfword).

Syntax

op{*XY*}{*cond*} *Rd*,*Rn*, *Rm* ; SMUL

op{*Y*}{*cond*} *Rd*. *Rn*, *Rm* ; SMULW

For SMUL{*XY*} only:

op

Is one of SMULBB, SMULBT, SMULTB, SMULTT:

SMUL{*XY*} Signed Multiply (halfwords)

X and *Y* specify which halfword of the source registers *Rn* and *Rm* is used as the first and second multiply operand. If *X* is B, then the bottom halfword, bits [15:0] of *Rn* is used. If *X* is T, then the top halfword, bits [31:16] of *Rn* is used. If *Y* is B, then the bottom halfword, bits [15:0], of *Rm* is used. If *Y* is T, then the top halfword, bits [31:16], of *Rm* is used.

SMULW{*Y*} Signed Multiply (word by halfword)

Y specifies which halfword of the source register *Rm* is used as the second multiply operand. If *Y* is B, then the bottom halfword (bits [15:0]) of *Rm* is used. If *Y* is T, then the top halfword (bits [31:16]) of *Rm* is used.

cond

Is an optional condition code.

Rd

Is the destination register.

Rn, *Rm*

Are registers holding the first and second operands.

Operation

The SMULBB, SMULTB, SMULBT and SMULTT instructions interprets the values from *Rn* and *Rm* as four signed 16-bit integers.

These instructions:

- Multiply the specified signed halfword, Top or Bottom, values from *Rn* and *Rm*.
- Write the 32-bit result of the multiplication in *Rd*.

The SMULWT and SMULWB instructions interprets the values from *Rn* as a 32-bit signed integer and *Rm* as two halfword 16-bit signed integers. These instructions:

- Multiply the first operand and the top, T suffix, or the bottom, B suffix, halfword of the second operand.
- Write the signed most significant 32 bits of the 48-bit result in the destination register.

Restrictions

In these instructions:

- Do not use SP and do not use PC.
- *RdHi* and *RdLo* must be different registers.

Example 3-34 Examples

```
SMULBT      R0, R4, R5      ; Multiplies the bottom halfword of R4 with the
                            ; top halfword of R5, multiplies results and
                            ; writes to R0.
SMULBB      R0, R4, R5      ; Multiplies the bottom halfword of R4 with the
                            ; bottom halfword of R5, multiplies results and
                            ; writes to R0.
SMULTT      R0, R4, R5      ; Multiplies the top halfword of R4 with the top
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                ; halfword of R5, multiplies results and writes
                                ; to R0.
SMULTB      R0, R4, R5          ; Multiplies the top halfword of R4 with the
                                ; bottom halfword of R5, multiplies results and
                                ; and writes to R0.
SMULWT      R4, R5, R3          ; Multiplies R5 with the top halfword of R3,
                                ; extracts top 32 bits and writes to R4.
SMULWB      R4, R5, R3          ; Multiplies R5 with the bottom halfword of R3,
                                ; extracts top 32 bits and writes to R4.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.15.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Multiply Long, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

Syntax

op{cond} RdLo, RdHi, Rn, Rm

Where:

op	Is one of:	
	`UMULL`	Unsigned Multiply Long.
	`UMLAL`	Unsigned Multiply, with Accumulate Long.
	`UMAAL`	Unsigned Long Multiply with Accumulate Accumulate.
	`SMULL`	Signed Multiply Long.
	`SMLAL`	Signed Multiply, with Accumulate Long.
cond	Is an optional condition code.	
RdHi, RdLo	Are the destination registers. For `UMLAL` and `SMLAL` they also hold the accumulating value of the lower and upper words respectively.	
Rn, Rm	Are registers holding the operands.	

Operation

The `UMULL` instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The `UMLAL` instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers, adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo*, and writes the result back to *RdHi* and *RdLo*.

The `UMAAL` instruction interprets the values from *Rn* and *Rm* as unsigned integers. It multiplies these integers, adds the unsigned 32-bit integer in *RdHi* to the 64-bit result of the multiplication, adds the unsigned 32-bit integer in *RdLo* to the 64-bit result of the addition, writes the top 32-bits of the result to *RdHi* and writes the lower 32-bits of the result to *RdLo*.

The `SMULL` instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers and places the least significant 32 bits of the result in *RdLo*, and the most significant 32 bits of the result in *RdHi*.

The `SMLAL` instruction interprets the values from *Rn* and *Rm* as two's complement signed integers. It multiplies these integers, adds the 64-bit result to the 64-bit signed integer contained in *RdHi* and *RdLo*, and writes the result back to *RdHi* and *RdLo*.

Restrictions

In these instructions:
- Do not use SP and do not use PC.
- *RdHi* and *RdLo* must be different registers.

Condition flags

These instructions do not affect the condition code flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-35 Examples

```
UMULL       R0, R4, R5, R6   ; Unsigned (R4,R0) = R5 × R6SMLAL       R4, R5, R3, R8   ;
Signed (R5,R4) = (R5,R4) + R3 × R8
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16 Saturating instructions

Reference material for the Cortex-M55 processor saturating instruction set.

3.16.1 List of saturating instructions

An alphabetically ordered list of the saturating instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-9 Saturating instructions

Mnemonic	Brief description	See
QADD	Saturating Add	*3.16.4 QADD and QSUB* on page 3-193
QASX	Saturating Add and Subtract with Exchange	*3.16.5 QASX and QSAX* on page 3-195
QDADD	Saturating Double and Add	*3.16.6 QDADD and QDSUB* on page 3-196
QDSUB	Saturating Double and Subtract	*3.16.6 QDADD and QDSUB* on page 3-196
QSAX	Saturating Subtract and Add with Exchange	*3.16.5 QASX and QSAX* on page 3-195
QSUB	Saturating Subtract	*3.16.4 QADD and QSUB* on page 3-193
QSUB16	Saturating Subtract 16	*3.16.4 QADD and QSUB* on page 3-193
SSAT	Signed Saturate	*3.16.2 SSAT and USAT* on page 3-191
SSAT16	Signed Saturate Halfword	*3.16.3 SSAT16 and USAT16* on page 3-192
UQADD16	Unsigned Saturating Add 16	*3.16.8 UQADD and UQSUB* on page 3-198
UQADD8	Unsigned Saturating Add 8	*3.16.8 UQADD and UQSUB* on page 3-198
UQASX	Unsigned Saturating Add and Subtract with Exchange	*3.16.7 UQASX and UQSAX* on page 3-197
UQSAX	Unsigned Saturating Subtract and Add with Exchange	*3.16.7 UQASX and UQSAX* on page 3-197
UQSUB16	Unsigned Saturating Subtract 16	*3.16.8 UQADD and UQSUB* on page 3-198
UQSUB8	Unsigned Saturating Subtract 8	*3.16.8 UQADD and UQSUB* on page 3-198
USAT	Unsigned Saturate	*3.16.2 SSAT and USAT* on page 3-191
USAT16	Unsigned Saturate Halfword	*3.16.3 SSAT16 and USAT16* on page 3-192

For signed *n*-bit saturation, this means that:

- If the value to be saturated is less than -2^{n-1}, the result returned is -2^{n-1}
- If the value to be saturated is greater than $2^{n-1}-1$, the result returned is $2^{n-1}-1$
- Otherwise, the result returned is the same as the value to be saturated.

For unsigned *n*-bit saturation, this means that:

- If the value to be saturated is less than 0, the result returned is 0
- If the value to be saturated is greater than $2^{n}-1$, the result returned is $2^{n}-1$
- Otherwise, the result returned is the same as the value to be saturated.

If the returned result is different from the value to be saturated, it is called *saturation*. If saturation occurs, the instruction sets the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. To clear the Q flag to 0, you must use the `MSR` instruction.

To read the state of the Q flag, use the `MRS` instruction.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.2 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

Syntax

op{cond} Rd, #n, Rm {, shift #s}

Where:

op
: Is one of:
  - `SSAT` Saturates a signed value to a signed range.
  - `USAT` Saturates a signed value to an unsigned range.

cond
: Is an optional condition code.

Rd
: Is the destination register.

n
: Specifies the bit position to saturate to:
  - *n* ranges from 1 to 32 for `SSAT`.
  - *n* ranges from 0 to 31 for `USAT`.

Rm
: Is the register containing the value to saturate.

shift #s
: Is an optional shift applied to *Rm* before saturating. It must be one of the following:
  - ASR #*s* where *s* is in the range 1-31.
  - LSL #*s* where *s* is in the range 0-31.

Operation

These instructions saturate to a signed or unsigned *n*-bit value.

The `SSAT` instruction applies the specified shift, then saturates to the signed range $-2^{n-1} \leq x \leq 2^{n-1}-1$.

The `USAT` instruction applies the specified shift, then saturates to the unsigned range $0 \leq x \leq 2^{n}-1$.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the Q flag to 1.

Example 3-36 Examples

```
SSAT    R7, #16, R7, LSL #4  ; Logical shift left value in R7 by 4, then
                             ; saturate it as a signed 16-bit value and
                             ; write it back to R7.
USATNE  R0, #7, R5           ; Conditionally saturate value in R5 as an
                             ; unsigned 7 bit value and write it to R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.3 SSAT16 and USAT16

Signed Saturate and Unsigned Saturate to any bit position for two halfwords.

Syntax

op{cond} Rd, #n, Rm

Where:

op	Is one of:	
	SSAT16	Saturates a signed halfword value to a signed range.
	USAT16	Saturates a signed halfword value to an unsigned range.
cond	Is an optional condition code.	
Rd	Is the destination register.	
n	Specifies the bit position to saturate to: • *n* ranges from 1 to 16 for SSAT. • *n* ranges from 0 to 15 for USAT.	
Rm	Is the register containing the values to saturate.	

Operation

The SSAT16 instruction:

1. Saturates two signed 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two signed 16-bit halfwords to the destination register.

The USAT16 instruction:

1. Saturates two unsigned 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two unsigned halfwords in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the Q flag to 1.

Example 3-37 Examples

```
SSAT16    R7, #9, R2       ; Saturates the top and bottom highwords of R2
                           ; as 9-bit values, writes to corresponding halfword
                           ; of R7.

USAT16NE  R0, #13, R5      ; Conditionally saturates the top and bottom
                           ; halfwords of R5 as 13-bit values, writes to
                           ; corresponding halfword of R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.4 QADD and QSUB

Saturating Add and Saturating Subtract, signed.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:
	`QADD` Saturating 32-bit add.
	`QADD8` Saturating four 8-bit integer additions.
	`QADD16` Saturating two 16-bit integer additions.
	`QSUB` Saturating 32-bit subtraction.
	`QSUB8` Saturating four 8-bit integer subtraction.
	`QSUB16` Saturating two 16-bit integer subtraction.
cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn, Rm	Are registers holding the first and second operands.

Operation

These instructions add or subtract two, four or eight values from the first and second operands and then writes a signed saturated value in the destination register.

The `QADD` and `QSUB` instructions apply the specified add or subtract, and then saturate the result to the signed range $-2^{n-1} \leq x \leq 2^{n-1}-1$, where x is given by the number of bits applied in the instruction, 32, 16 or 8.

If the returned result is different from the value to be saturated, it is called *saturation*. If saturation occurs, the `QADD` and `QSUB` instructions set the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. The 8-bit and 16-bit `QADD` and `QSUB` instructions always leave the Q flag unchanged.

To clear the Q flag to 0, you must use the `MSR` instruction.

To read the state of the Q flag, use the `MRS` instruction.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

If saturation occurs, the `QADD` and `QSUB` instructions set the Q flag to 1.

Example 3-38 Examples

```
QADD16   R7, R4, R2  ; Adds halfwords of R4 with corresponding halfword of
                     ; R2, saturates to 16 bits and writes to corresponding
                     ; halfword of R7.

QADD8    R3, R1, R6  ; Adds bytes of R1 to the corresponding bytes of R6,
                     ; saturates to 8 bits and writes to corresponding byte of
                     ; R3.

QSUB16   R4, R2, R3  ; Subtracts halfwords of R3 from corresponding halfword
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                     ; of R2, saturates to 16 bits, writes to corresponding
                     ; halfword of R4.

QSUB8    R4, R2, R5  ; Subtracts bytes of R5 from the corresponding byte in
                     ; R2, saturates to 8 bits, writes to corresponding byte of
                     ; R4.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.5 QASX and QSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, signed.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	`QASX`	Add and Subtract with Exchange and Saturate.
	`QSAX`	Subtract and Add with Exchange and Saturate.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn, *Rm*	Are registers holding the first and second operands.	

Operation

The `QASX` instruction:

1. Adds the top halfword of the source operand with the bottom halfword of the second operand.
2. Subtracts the top halfword of the second operand from the bottom highword of the first operand.
3. Saturates the result of the subtraction and writes a 16-bit signed integer in the range $-215 \leq x \leq 215 - 1$, where x equals 16, to the bottom halfword of the destination register.
4. Saturates the results of the sum and writes a 16-bit signed integer in the range $-215 \leq x \leq 215 - 1$, where x equals 16, to the top halfword of the destination register.

The `QSAX` instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Adds the bottom halfword of the source operand with the top halfword of the second operand.
3. Saturates the results of the sum and writes a 16-bit signed integer in the range $-215 \leq x \leq 215 - 1$, where x equals 16, to the bottom halfword of the destination register.
4. Saturates the result of the subtraction and writes a 16-bit signed integer in the range $-215 \leq x \leq 215 - 1$, where x equals 16, to the top halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Example 3-39 Examples

```
QASX    R7, R4, R2     ; Adds top halfword of R4 to bottom halfword of R2,
                       ; saturates to 16 bits, writes to top halfword of R7
                       ; Subtracts top highword of R2 from bottom halfword of
                       ; R4, saturates to 16 bits and writes to bottom halfword
                       ; of R7

QSAX    R0, R3, R5     ; Subtracts bottom halfword of R5 from top halfword of
                       ; R3, saturates to 16 bits, writes to top halfword of R0
                       ; Adds bottom halfword of R3 to top halfword of R5,
                       ; saturates to 16 bits, writes to bottom halfword of R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.6 QDADD and QDSUB

Saturating Double and Add and Saturating Double and Subtract, signed.

Syntax

op{cond} {Rd}, Rm, Rn

Where:

op	Is one of: `QDADD` Saturating Double and Add. `QDSUB` Saturating Double and Subtract.
cond	Is an optional condition code.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rm, Rn	Are registers holding the first and second operands.

Operation

The `QDADD` instruction:

- Doubles the second operand value.
- Adds the result of the doubling to the signed saturated value in the first operand.
- Writes the result to the destination register.

The `QDSUB` instruction:

- Doubles the second operand value.
- Subtracts the doubled value from the signed saturated value in the first operand.
- Writes the result to the destination register.

Both the doubling and the addition or subtraction have their results saturated to the 32-bit signed integer range $–231 \leq x \leq 231– 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

Restrictions

Do not use SP and do not use PC.

Condition flags

If saturation occurs, these instructions set the Q flag to 1.

Example 3-40 Examples

```
QDADD     R7, R4, R2          ; Doubles and saturates R4 to 32 bits, adds R2,
                              ; saturates to 32 bits, writes to R7

QDSUB     R0, R3, R5          ; Subtracts R3 doubled and saturated to 32 bits
                              ; from R5, saturates to 32 bits, writes to R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.7 UQASX and UQSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, unsigned.

Syntax

`op{cond} {Rd,} Rn, Rm`

Where:

type
: Is one of:

UQASX	Add and Subtract with Exchange and Saturate.
UQSAX	Subtract and Add with Exchange and Saturate.

cond
: Is an optional condition code.

Rd
: Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn, Rm
: Are registers holding the first and second operands.

Operation

The `UQASX` instruction:

1. Adds the bottom halfword of the source operand with the top halfword of the second operand.
2. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
3. Saturates the results of the sum and writes a 16-bit unsigned integer in the range $0 \leq x \leq 216 - 1$, where x equals 16, to the top halfword of the destination register.
4. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range $0 \leq x \leq 216 - 1$, where x equals 16, to the bottom halfword of the destination register.

The `UQSAX` instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Adds the bottom halfword of the first operand with the top halfword of the second operand.
3. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range $0 \leq x \leq 216 - 1$, where x equals 16, to the top halfword of the destination register.
4. Saturates the results of the addition and writes a 16-bit unsigned integer in the range $0 \leq x \leq 216 - 1$, where x equals 16, to the bottom halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Example 3-41 Examples

```
UQASX   R7, R4, R2  ; Adds top halfword of R4 with bottom halfword of R2,
                    ; saturates to 16 bits, writes to top halfword of R7
                    ; Subtracts top halfword of R2 from bottom halfword of
                    ; R4, saturates to 16 bits, writes to bottom halfword of R7
UQSAX   R0, R3, R5  ; Subtracts bottom halfword of R5 from top halfword of R3,
                    ; saturates to 16 bits, writes to top halfword of R0
                    ; Adds bottom halfword of R4 to top halfword of R5
                    ; saturates to 16 bits, writes to bottom halfword of R0.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.16.8 UQADD and UQSUB

Saturating Add and Saturating Subtract Unsigned.

Syntax

op{cond} {Rd,} Rn, Rm

Where:

op	Is one of:	
	UQADD8	Saturating four unsigned 8-bit integer additions.
	UQADD16	Saturating two unsigned 16-bit integer additions.
	UQSUB8	Saturating four unsigned 8-bit integer subtractions.
	UQSUB16	Saturating two unsigned 16-bit integer subtractions.
cond	Is an optional condition code.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn, Rm	Are registers holding the first and second operands.	

Operation

These instructions add or subtract two or four values and then writes an unsigned saturated value in the destination register.

The UQADD16 instruction:

- Adds the respective top and bottom halfwords of the first and second operands.
- Saturates the result of the additions for each halfword in the destination register to the unsigned range $0 \leq x \leq 2^{16}-1$, where x is 16.

The UQADD8 instruction:

- Adds each respective byte of the first and second operands.
- Saturates the result of the addition for each byte in the destination register to the unsigned range $0 \leq x \leq 2^{8}-1$, where x is 8.

The UQSUB16 instruction:

- Subtracts both halfwords of the second operand from the respective halfwords of the first operand.
- Saturates the result of the differences in the destination register to the unsigned range $0 \leq x \leq 2^{16}-1$, where x is 16.

The UQSUB8 instructions:

- Subtracts the respective bytes of the second operand from the respective bytes of the first operand.
- Saturates the results of the differences for each byte in the destination register to the unsigned range $0 \leq x \leq 2^{8}-1$, where x is 8.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-42 Examples

```
UQADD16  R7, R4, R2   ; Adds halfwords in R4 to corresponding halfword in R2,
                      ; saturates to 16 bits, writes to corresponding halfword
                      ; of R7
UQADD8   R4, R2, R5   ; Adds bytes of R2 to corresponding byte of R5, saturates
                      ; to 8 bits, writes to corresponding bytes of R4
UQSUB16  R6, R3, R0   ; Subtracts halfwords in R0 from corresponding halfword
                      ; in R3, saturates to 16 bits, writes to corresponding
                      ; halfword in R6
UQSUB8   R1, R5, R6   ; Subtracts bytes in R6 from corresponding byte of R5,
                      ; saturates to 8 bits, writes to corresponding byte of R1.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.17 Packing and unpacking instructions

Reference material for the Cortex-M55 processor packing and unpacking instruction set.

3.17.1 List of packing and unpacking instructions

An alphabetically ordered list of the packing and unpacking instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-10 Packing and unpacking instructions

Mnemonic	Brief description	See
PKH	Pack Halfword	*3.17.2 PKHBT and PKHTB* on page 3-201
SXTAB	Extend 8 bits to 32 and add	*3.17.3 SXTA and UXTA* on page 3-203
SXTAB16	Dual extend 8 bits to 16 and add	*3.17.3 SXTA and UXTA* on page 3-203
SXTAH	Extend 16 bits to 32 and add	*3.17.3 SXTA and UXTA* on page 3-203
SXTB	Sign extend a byte	*3.17.4 SXT and UXT* on page 3-205
SXTB16	Dual extend 8 bits to 16 and add	*3.17.4 SXT and UXT* on page 3-205
SXTH	Sign extend a halfword	*3.17.4 SXT and UXT* on page 3-205
UXTAB	Extend 8 bits to 32 and add	*3.17.3 SXTA and UXTA* on page 3-203
UXTAB16	Dual extend 8 bits to 16 and add	*3.17.3 SXTA and UXTA* on page 3-203
UXTAH	Extend 16 bits to 32 and add	*3.17.3 SXTA and UXTA* on page 3-203
UXTB	Zero extend a byte	*3.17.4 SXT and UXT* on page 3-205
UXTB16	Dual zero extend 8 bits to 16 and add	*3.17.4 SXT and UXT* on page 3-205
UXTH	Zero extend a halfword	*3.17.4 SXT and UXT* on page 3-205

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.17.2 PKHBT and PKHTB

Pack Halfword.

Syntax

`op{cond} {Rd}, Rn, Rm {, LSL #imm} ;PKHBT`

`op{cond} {Rd}, Rn, Rm {, ASR #imm} ;PKHTB`

Where:

op
: Is one of:

`PKHBT`	Pack Halfword, bottom and top with shift.
`PKHTB`	Pack Halfword, top and bottom with shift.

cond
: Is an optional condition code.

Rd
: Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn
: Is the first operand register.

Rm
: Is the second operand register holding the value to be optionally shifted.

imm
: Is the shift length. The type of shift length depends on the instruction:For `PKHBT`:

For `PKHTB`:

`LSL`	A left shift with a shift length from 1 to 31, 0 means no shift.
`ASR`	An arithmetic shift right with a shift length from 1 to 32, a shift of 32-bits is encoded as `0b00000`.

Operation

`The PKHBT` instruction:

1. Writes the value of the bottom halfword of the first operand to the bottom halfword of the destination register.
2. If shifted, the shifted value of the second operand is written to the top halfword of the destination register.

The `PKHTB` instruction:

1. Writes the value of the top halfword of the first operand to the top halfword of the destination register.
2. If shifted, the shifted value of the second operand is written to the bottom halfword of the destination register.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-43 Examples

```
PKHBT   R3, R4, R5 LSL #0  ; Writes bottom halfword of R4 to bottom halfword of
                           ; R3, writes top halfword of R5, unshifted, to top
                           ; halfword of R3

PKHTB   R4, R0, R2 ASR #1  ; Writes R2 shifted right by 1 bit to bottom halfword
                           ; of R4, and writes top halfword of R0 to top
                           ; halfword of R4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.17.3 SXTA and UXTA

Signed and Unsigned Extend and Add.

Syntax

`op{cond} {Rd,} Rn, Rm {, ROR #n}`

Where:

op
: Is one of:

`SXTAB`	Sign extends an 8-bit value to a 32-bit value and add.
`SXTAH`	Sign extends a 16-bit value to a 32-bit value and add.
`SXTAB16`	Sign extends two 8-bit values to two 16-bit values and add.
`UXTAB`	Zero extends an 8-bit value to a 32-bit value and add.
`UXTAH`	Zero extends a 16-bit value to a 32-bit value and add.
`UXTAB16`	Zero extends two 8-bit values to two 16-bit values and add.

cond
: Is an optional condition code.

Rd
: Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn
: Is the first operand register.

Rm
: Is the register holding the value to rotate and extend.

`ROR #n`
: Is one of:

`ROR #8`	Value from *Rm* is rotated right 8 bits.
`ROR #16`	Value from *Rm* is rotated right 16 bits.
`ROR #24`	Value from *Rm* is rotated right 24 bits.

If ROR #*n* is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from *Rm* right by 0, 8, 16 or 24 bits.
2. Extract bits from the resulting value:
   - `SXTAB` extracts bits[7:0] from *Rm* and sign extends to 32 bits.
   - `UXTAB` extracts bits[7:0] from *Rm* and zero extends to 32 bits.
   - `SXTAH` extracts bits[15:0] from *Rm* and sign extends to 32 bits.
   - `UXTAH` extracts bits[15:0] from *Rm* and zero extends to 32 bits.
   - `SXTAB16` extracts bits[7:0] from *Rm* and sign extends to 16 bits, and extracts bits [23:16] from *Rm* and sign extends to 16 bits.
   - `UXTAB16` extracts bits[7:0] from *Rm* and zero extends to 16 bits, and extracts bits [23:16] from *Rm* and zero extends to 16 bits.
3. Adds the signed or zero extended value to the word or corresponding halfword of *Rn* and writes the result in *Rd*.

Restrictions

Do not use SP and do not use PC.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Condition flags

These instructions do not affect the flags.

Example 3-44 Examples

```
SXTAH  R4, R8, R6, ROR #16 ; Rotates R6 right by 16 bits, obtains bottom
                           ; halfword, sign extends to 32 bits, adds R8,and
                           ; writes to R4
UXTAB  R3, R4, R10         ; Extracts bottom byte of R10 and zero extends to 32
                           ; bits, adds R4, and writes to R3.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.17.4 SXT and UXT

Sign extend and Zero extend.

Syntax

`SXTop{cond} Rd, Rn {, ROR #n}`

`UXTop{cond} Rd, Rn {, ROR #n}`

Where:

op
Is one of:

SXTB	Sign extends an 8-bit value to a 32-bit value.
SXTH	Sign extends a 16-bit value to a 32-bit value.
SXTB16	Sign extends two 8-bit values to two 16-bit values.
UXTB	Zero extends an 8-bit value to a 32-bit value.
UXTH	Zero extends a 16-bit value to a 32-bit value.
UXTB16	Zero extends two 8-bit values to two 16-bit values.

cond
Is an optional condition code.

Rd
Is the destination register.

Rn
Is the register holding the value to extend.

ROR #*n*
Is one of:

ROR #8	Value from *Rn* is rotated right 8 bits.
ROR #16	Value from *Rn* is rotated right 16 bits.
ROR #24	Value from *Rn* is rotated right 24 bits.

If ROR #*n* is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from *Rn* right by 0, 8, 16 or 24 bits.
2. Extract bits from the resulting value:
   - SXTB extracts bits[7:0] and sign extends to 32 bits.
   - UXTB extracts bits[7:0] and zero extends to 32 bits.
   - SXTH extracts bits[15:0] and sign extends to 32 bits.
   - UXTH extracts bits[15:0] and zero extends to 32 bits.
   - SXTB16 extracts bits[7:0] and sign extends to 16 bits, and extracts bits [23:16] and sign extends to 16 bits.
   - UXTB16 extracts bits[7:0] and zero extends to 16 bits, and extracts bits [23:16] and zero extends to 16 bits.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-45 Examples

```
SXTH  R4, R6, ROR #16  ; Rotate R6 right by 16 bits, then obtain the lower
                       ; halfword of the result and then sign extend to
                       ; 32 bits and write the result to R4.
UXTB  R3, R10          ; Extract lowest byte of the value in R10 and zero
                       ; extend it, and write the result to R3.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.18 Bit field instructions

Reference material for the Cortex-M55 processor bit field instruction set.

3.18.1 List of bit field instructions

An alphabetically ordered list of the bit field instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-11 Bit field instructions

Mnemonic	Brief description	See
`BFC`	Bit Field Clear	*3.18.2 BFC and BFI* on page 3-208
`BFI`	Bit Field Insert	*3.18.2 BFC and BFI* on page 3-208
`SBFX`	Signed Bit Field Extract	*3.18.3 SBFX and UBFX* on page 3-209
`UBFX`	Unsigned Bit Field Extract	*3.18.3 SBFX and UBFX* on page 3-209

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.18.2 BFC and BFI

Bit Field Clear and Bit Field Insert.

Syntax

`BFC{cond} Rd, #lsb, #width`

`BFI{cond} Rd, Rn, #lsb, #width`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
Rn	Is the source register.
lsb	Is the position of the least significant bit of the bit field. *lsb* must be in the range 0-31.
width	Is the width of the bit field and must be in the range 1-32−*lsb*.

Operation

`BFC` clears a bit field in a register. It clears *width* bits in *Rd*, starting at the low bit position *lsb*. Other bits in *Rd* are unchanged.

`BFI` copies a bit field into one register from another register. It replaces *width* bits in *Rd* starting at the low bit position *lsb*, with *width* bits from *Rn* starting at bit[0]. Other bits in *Rd* are unchanged.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Example 3-46 Examples

```
BFC   R4, #8, #12       ; Clear bit 8 to bit 19 (12 bits) of R4 to 0
BFI   R9, R2, #8, #12   ; Replace bit 8 to bit 19 (12 bits) of R9 with
                        ; bit 0 to bit 11 from R2.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.18.3 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Syntax

`SBFX{cond} Rd, Rn, #lsb, #width`

`UBFX{cond} Rd, Rn, #lsb, #width`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
Rn	Is the source register.
lsb	Is the position of the least significant bit of the bit field. *lsb* must be in the range 0-31.
width	Is the width of the bit field and must be in the range 1-32−*lsb*.

Operation

`SBFX` extracts a bit field from one register, sign extends it to 32 bits, and writes the result to the destination register.

`UBFX` extracts a bit field from one register, zero extends it to 32 bits, and writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Example 3-47 Examples

```
SBFX  R0, R1, #20, #4  ; Extract bit 20 to bit 23 (4 bits) from R1 and sign
                       ; extend to 32 bits and then write the result to R0.
UBFX  R8, R11, #9, #10 ; Extract bit 9 to bit 18 (10 bits) from R11 and zero
                       ; extend to 32 bits and then write the result to R8.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19 Branch and control instructions

Reference material for the Cortex-M55 processor branch and control instruction set.

3.19.1 List of branch and control instructions

An alphabetically ordered list of the branch and control instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-12 Branch and control instructions

Mnemonic	Brief description	See
`B`	Branch	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BL`	Branch with Link	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BLX`	Branch indirect with Link	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BLXNS`	Branch indirect with Link, Non-secure	*3.19.3 BXNS and BLXNS* on page 3-213
`BX`	Branch indirect	*3.19.2 B, BL, BX, and BLX* on page 3-211
`BXNS`	Branch indirect, Non-secure	*3.19.3 BXNS and BLXNS* on page 3-213
`CBNZ`	Compare and Branch if Non Zero	*3.19.4 CBZ and CBNZ* on page 3-214
`CBZ`	Compare and Branch if Zero	*3.19.4 CBZ and CBNZ* on page 3-214
`IT`	If-Then	*3.19.5 IT* on page 3-215
`TBB`	Table Branch Byte	*3.19.6 TBB and TBH* on page 3-217
`TBH`	Table Branch Halfword	*3.19.6 TBB and TBH* on page 3-217

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19.2 B, BL, BX, and BLX

Branch instructions.

Syntax

`B{cond} label`

`BL label`

`BX Rm`

`BLX Rm`

Where:

cond Is an optional condition code.

label Is a PC-relative expression.

Rm Is a register providing the address to branch to.

Operation

All these instructions cause a branch to the address indicated by *label* or contained in the register specified by *Rm*. In addition:

- The `BL` and `BLX` instructions write the address of the next instruction to LR, the link register R14.
- The `BX` and `BLX` instructions result in a UsageFault exception if bit[0] of *Rm* is 0.

`BL` and `BLX` instructions also set bit[0] of the LR to 1. This ensures that the value is suitable for use by a subsequent `POP {PC}` or `BX` instruction to perform a successful return branch.

The following table shows the ranges for the various branch instructions.

Table 3-13 Branch ranges

Instruction	Branch range
`B label`	−16MB to +16MB.
`Bcond label`	−1MB to +1MB
`BL label`	−16MB to +16MB.
`BX Rm`	Any value in register.
`BLX Rm`	Any value in register.

Restrictions

In these instructions:

- Do not use SP or PC in the `BX` or `BLX` instruction.
- For `BX` and `BLX`, bit[0] of *Rm* must be 1 for correct execution. Bit[0] is used to update the EPSR T-bit and is discarded from the target address.

Note

`Bcond` is the only conditional instruction on the processor.

`BX` can be used an Exception or Function return.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Examples

```
    B      loopA  ; Branch to loopA
    BL     funC   ; Branch with link (Call) to function funC, return address
                  ; stored in LR
    BX     LR     ; Return from function call if LR contains a FUNC_RETURN value.
    BLX    R0     ; Branch with link and exchange (Call) to a address stored
                  ; in R0
    BEQ    labelD ; Conditionally branch to labelD if last flag setting
                  ; instruction set the Z flag, else do not branch.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19.3 BXNS and BLXNS

Branch and Exchange Non-secure and Branch with Link and Exchange Non-secure.

Syntax

`BXNS <Rm>`

`BLXNS <Rm>`

Where:

Rm Is a register containing an address to branch to.

Operation

The `BLXNS` instruction calls a subroutine at an address contained in *Rm* and conditionally causes a transition from the Secure to the Non-secure state.

For both `BXNS` and `BLXNS`, Rm[0] indicates a transition to Non-secure state if value is 0, otherwise the target state remains Secure. If transitioning to Non-secure, `BLXNS` pushes the return address and partial PSR to the Secure stack and assigns R14 to a FNC_RETURN value.

These instructions are available for Secure state only. When the processor is in Non-secure state, these instructions are UNDEFINED and triggers a UsageFault if executed.

Restrictions

PC and SP cannot be used for *Rm*.

Condition flags

These instructions do not change the flags.

Examples

```
LDR r0, =non_secure_function
MOVS r1, #1
BICS r0, r1 # Clear bit 0 of address in r0
BLXNS r0 ; Call Non-secure function. This sets r14 to FUNC_RETURN value
```

Note

For information about how to build a Secure image that uses a previously generated import library, see the *Arm® Compiler Software Development Guide*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19.4 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

Syntax

op{cond} Rn, label

Where:

cond	Is an optional condition code.
Rn	Is the register holding the operand.
label	Is the branch destination.

Operation

Use the `CBZ` or `CBNZ` instructions to avoid changing the condition code flags and to reduce the number of instructions.

`CBZ Rn, label` does not change condition flags but is otherwise equivalent to:

```
CMP     Rn, #0    BEQ     label
```

`CBNZ Rn, label` does not change condition flags but is otherwise equivalent to:

```
CMP     Rn, #0    BNE     label
```

Restrictions

The restrictions are:
- *Rn* must be in the range of `R0-R7`.
- The branch destination must be within 4 to 130 bytes after the instruction.
- These instructions must not be used inside an IT block.

Condition flags

These instructions do not change the flags.

Example 3-48 Examples

```
CBZ    R5, target ; Forward branch if R5 is zero
CBNZ   R0, target ; Forward branch if R0 is not zero
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19.5 IT

If-Then condition instruction.

Syntax

`IT{x{y{z}}} cond`

Where:

x	specifies the condition switch for the second instruction in the IT block.
y	Specifies the condition switch for the third instruction in the IT block.
z	Specifies the condition switch for the fourth instruction in the IT block.
cond	Specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

`T`	Then. Applies the condition *cond* to the instruction.
`E`	Else. Applies the inverse condition of *cond* to the instruction.

Note

It is possible to use `AL` (the *always* condition) for *cond* in an `IT` instruction. If this is done, all of the instructions in the IT block must be unconditional, and each of *x*, *y*, and *z* must be `T` or omitted but not `E`.

Operation

The `IT` instruction makes up to four following instructions conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. The conditional instructions following the `IT` instruction form the *IT block*.

The instructions in the IT block, including any branches, must specify the condition in the `{cond}` part of their syntax.

Note

Your assembler might be able to generate the required `IT` instructions for conditional instructions automatically, so that you do not have to write them yourself. See your assembler documentation for details.

A `BKPT` instruction in an IT block is always executed, even if its condition fails.

Exceptions can be taken between an `IT` instruction and the corresponding IT block, or within an IT block. Such an exception results in entry to the appropriate exception handler, with suitable return information in LR and stacked PSR.

Instructions designed for use for exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction is permitted to branch to an instruction in an IT block.

Restrictions

The following instructions are not permitted in an IT block:

- `IT`.
- `CBZ` and `CBNZ`.
- `CPSID` and `CPSIE.`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. These are:
  - `ADD PC, PC, Rm`.
  - `MOV PC, Rm`.
  - `B`, `BL`, `BX`, `BLX`.
  - Any `LDM`, `LDR`, or `POP` instruction that writes to the PC.
  - `TBB` and `TBH`.
- Do not branch to any instruction inside an IT block, except when returning from an exception handler.
- All conditional instructions except B*cond* must be inside an IT block. B*cond* can be either outside or inside an IT block but has a larger branch range if it is inside one.
- Each instruction inside the IT block must specify a condition code suffix that is either the same or logical inverse as for the other instructions in the block.

Note

Your assembler might place extra restrictions on the use of IT blocks, such as prohibiting the use of assembler directives within them.

Condition flags

This instruction does not change the flags.

Example 3-49 Examples

```
ITTE   NE           ; Next 3 instructions are conditional
ANDNE  R0, R0, R1   ; ANDNE does not update condition flags
ADDSNE R2, R2, #1   ; ADDSNE updates condition flags
MOVEQ  R2, R3       ; Conditional move
CMP    R0, #9       ; Convert R0 hex value (0 to 15) into ASCII
                    ; ('0'-'9', 'A'-'F')
ITE    GT           ; Next 2 instructions are conditional
ADDGT  R1, R0, #55  ; Convert 0xA -> 'A'
ADDLE  R1, R0, #48  ; Convert 0x0 -> '0'
IT     GT           ; IT block with only one conditional instruction
ADDGT  R1, R1, #1   ; Increment R1 conditionally ITTEE  EQ
                    ; Next 4 instructions are conditional
MOVEQ  R0, R1       ; Conditional move
ADDEQ  R2, R2, #10  ; Conditional add
ANDNE  R3, R3, #1   ; Conditional AND
BNE.W  dloop        ; Branch instruction can only be used in the last
                    ; instruction of an IT block
IT     NE           ; Next instruction is conditional
ADD    R0, R0, R1   ; Syntax error: no condition code used in IT block
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.19.6 TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

`TBB [Rn, Rm]`

`TBH [Rn, Rm, LSL #1]`

Where:

Rn
Is the register containing the address of the table of branch lengths.

If *Rn* is PC, then the address of the table is the address of the byte immediately following the `TBB` or `TBH` instruction.

Rm
Is the index register. This contains an index into the table. For halfword tables, `LSL #1` doubles the value in *Rm* to form the right offset into the table.

Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets for `TBB`, or halfword offsets for `TBH`. *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. For `TBB` the branch offset is the unsigned value of the byte returned from the table, and for `TBH` the branch offset is twice the unsigned value of the halfword returned from the table. The branch occurs to the address at that offset from the address of the byte immediately after the `TBB` or `TBH` instruction.

Restrictions

The restrictions are:

- *Rn* must not be SP.
- *Rm* must not be SP and must not be PC.
- When any of these instructions is used inside an IT block, it must be the last instruction of the IT block.

Condition flags

These instructions do not change the flags.

Example 3-50 Examples

```
ADR.W  R0, BranchTable_Byte
TBB   [R0, R1]        ; R1 is the index, R0 is the base address of the
                      ; branch table
Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable_Byte
    DCB    0                  ; Case1 offset calculation
    DCB    ((Case2-Case1)/2)  ; Case2 offset calculation
    DCB    ((Case3-Case1)/2)  ; Case3 offset calculation
    TBH    [PC, R1, LSL #1]   ; R1 is the index, PC is used as base of the
                              ; branch table
BranchTable_H
    DCW    ((CaseA - BranchTable_H)/2)  ; CaseA offset calculation
    DCW    ((CaseB - BranchTable_H)/2)  ; CaseB offset calculation
    DCW    ((CaseC - BranchTable_H)/2)  ; CaseC offset calculation
CaseA
; an instruction sequence follows
CaseB
; an instruction sequence follows
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CaseC
; an instruction sequence follows
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20 Floating-point instructions

Reference material for the Cortex-M55 processor floating-point instruction set that the FPU uses.

3.20.1 List of floating-point instructions

An alphabetically ordered list of the floating-point instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Note

These instructions are only available if the FPU is included, and enabled, in the system.

Table 3-14 Floating-point instructions

Mnemonic	Brief description	See
FLDMDBX	FLDMX (Decrement Before) loads multiple extension registers from consecutive memory locations	*3.20.2 FLDMDBX, FLDMIAX* on page 3-222
FLDMIAX	FLDMX (Increment After) loads multiple extension registers from consecutive memory locations	*3.20.2 FLDMDBX, FLDMIAX* on page 3-222
FSTMDBX	FSTMX (Decrement Before) stores multiple extension registers to consecutive memory locations	*3.20.3 FSTMDBX, FSTMIAX* on page 3-223
FSTMIAX	FSTMX (Increment After) stores multiple extension registers to consecutive memory locations	*3.20.3 FSTMDBX, FSTMIAX* on page 3-223
VABS	Floating-point Absolute	*3.20.4 VABS* on page 3-224
VADD	Floating-point Add	*3.20.5 VADD* on page 3-225
VCMP	Compare two floating-point registers, or one floating-point register and zero	*3.20.6 VCMP and VCMPE* on page 3-226
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	*3.20.6 VCMP and VCMPE* on page 3-226
VCVT	Convert between floating-point and integer	*3.20.7 VCVT and VCVTR between floating-point and integer* on page 3-227
VCVT	Convert between floating-point and fixed point	*3.20.8 VCVT between floating-point and fixed-point* on page 3-228
VCVTA, VCVTN, VCVTP, VCVTM	Float to integer conversion with directed rounding	*3.20.36 VCVTA, VCVTM VCVTN, and VCVTP* on page 3-256
VCVTB	Converts half-precision value to single-precision	*3.20.37 VCVTB and VCVTT* on page 3-257
VCVTR	Convert between floating-point and integer with rounding	*3.20.7 VCVT and VCVTR between floating-point and integer* on page 3-227
VCVTT	Converts single-precision register to half-precision	*3.20.37 VCVTB and VCVTT* on page 3-257
VDIV	Floating-point Divide	*3.20.9 VDIV* on page 3-229

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-14 Floating-point instructions (continued)

Mnemonic	Brief description	See
VFMA	Floating-point Fused Multiply Accumulate	*3.20.10 VFMA and VFMS* on page 3-230
VFMS	Floating-point Fused Multiply Subtract	*3.20.10 VFMA and VFMS* on page 3-230
VFNMA	Floating-point Fused Negate Multiply Accumulate	*3.20.11 VFNMA and VFNMS* on page 3-231
VFNMS	Floating-point Fused Negate Multiply Subtract	*3.20.11 VFNMA and VFNMS* on page 3-231
VLDM	Load Multiple extension registers	*3.20.12 VLDM* on page 3-232
VLDR	Loads an extension register from memory	*3.20.13 VLDR* on page 3-233
VMAXNM, VMINNM	Maximum, Minimum with IEEE754-2008 NaN handling	*3.20.38 VMAXNM and VMINNM* on page 3-258
VMLA	Floating-point Multiply Accumulate	*3.20.16 VMLA and VMLS* on page 3-236
VMLS	Floating-point Multiply Subtract	*3.20.16 VMLA and VMLS* on page 3-236
VMOV	Floating-point Move Immediate	*3.20.17 VMOV Immediate* on page 3-237
VMOV	Floating-point Move Register	*3.20.18 VMOV Register* on page 3-238
VMOV	Copy Arm core register to single-precision	*3.20.20 VMOV core register to single-precision* on page 3-240
VMOV	Copy 2 Arm core registers to 2 single-precision	*3.20.21 VMOV two core registers to two single-precision registers* on page 3-241
VMOV	Copies between Arm core register to scalar	*3.20.23 VMOV core register to scalar* on page 3-243
VMOV	Copies between Scalar to Arm core register	*3.20.19 VMOV scalar to core register* on page 3-239
VMRS	Move to Arm core register from floating-point System Register	*3.20.24 VMRS* on page 3-244
VMSR	Move to floating-point System Register from Arm Core register	*3.20.25 VMSR* on page 3-245
VMUL	Multiply floating-point	*3.20.26 VMUL* on page 3-246
VNEG	Floating-point negate	*3.20.27 VNEG* on page 3-247
VNMLA	Floating-point multiply and add	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248
VNMLS	Floating-point multiply and subtract	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248
VNMUL	Floating-point multiply	*3.20.28 VNMLA, VNMLS and VNMUL* on page 3-248
VPOP	Pop extension registers	*3.20.29 VPOP* on page 3-249
VPUSH	Push extension registers	*3.20.30 VPUSH* on page 3-250
VRINTA, VRINTN, VRINTP, VRINTM	Float to integer (in floating-point format) conversion with directed rounding	*3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-260
VRINTR, VRINTX	Float to integer (in floating-point format) conversion	*3.20.39 VRINTR and VRINTX* on page 3-259
VSEL	Select register, alternative to a pair of conditional VMOV	*3.20.35 VSEL* on page 3-255
VSQRT	Floating-point square root	*3.20.31 VSQRT* on page 3-251
VSTM	Store Multiple extension registers	*3.20.32 VSTM* on page 3-252

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-14 Floating-point instructions (continued)

Mnemonic	Brief description	See
VSTR	Stores an extension register to memory	*3.20.33 VSTR* on page 3-253
VSUB	Floating-point Subtract	*3.20.34 VSUB* on page 3-254

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.2 FLDMDBX, FLDMIAX

`FLDMX` (Decrement Before, Increment After) loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

Syntax

`FLDMDBX{cond} Rn!, dreglist`

`FLDMIAX{cond} Rn{!}, dreglist`

Where:

cond Is an optional condition code.

Rn Is the base register. If write-back is not specified, the PC can be used.

! Specifies base register write-back.

dreglist Is the list of consecutively numbered 64-bit SIMD and FP registers to be transferred. 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

`FLDMX` loads multiple SIMD and FP registers from consecutive locations in the Advanced SIMD and floating-point register file using an address from a general-purpose register.

Arm deprecates use of `FLDMDBX` and `FLDMIAX`, except for disassembly purposes, and reassembly of disassembled code.

Depending on settings in the *CPACR* and *NSACR* and the Security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.3 FSTMDBX, FSTMIAX

`FSTMX` (Decrement Before, Increment After) stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

Syntax

`FSTMDBX{c}{q} Rn!, dreglist`

`FSTMIAX{c}{q} Rn{!}, dreglist`

Where:

cond Is an optional condition code.

Rn Is the base register. If write-back is not specified, the PC can be used. However, Arm deprecates use of the PC.

! Specifies base register write-back.

dreglist Is the list FP registers to be transferred. 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

`FSTMX` stores multiple SIMD and FP registers from the Advanced SIMD and floating-point register file to consecutive 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.

Depending on settings in the *CPACR*, *NSACR*, and *FPEXC* Registers, and the security state and mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.4 VABS

Floating-point Absolute.

Syntax

```
VABS{cond}.F16 Sd, Sm
```

```
VABS{cond}.F32 Sd, Sm
```

```
VABS{cond}.F64 Sd, Sm
```

Where:

cond	Is an optional condition code.
Sd, Sm	Are the destination floating-point value and the operand floating-point value.

Operation

This instruction:

1. Takes the absolute value of the operand floating-point register.
2. Places the results in the destination floating-point register.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Example 3-51 Examples

```
VABS.F32 S4, S6
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.5 VADD

Floating-point Add.

Syntax

```
VADD{cond}.F32 {Sd,} Sn, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sn, Sm	Are the operand floating-point values.

Operation

This instruction:

1. Adds the values in the two floating-point operand registers.
2. Places the results in the destination floating-point register.
3. the results in the destination floating-point register.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Example 3-52 Examples

```
VADD.F32 S4, S6, S7
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.6 VCMP and VCMPE

Compares two floating-point registers, or one floating-point register and zero.

Syntax

`VCMP{E}{cond}.F32 Sd, Sm|#0.0`

`VCMP{E}{cond}.F32 Sd, #0.0`

Where:

cond	Is an optional condition code.
E	If present, any `NaN` operand causes an `Invalid Operation` exception. Otherwise, only a signaling `NaN` causes the exception.
Sd	Is the floating-point operand to compare.
Sm|Dm	Is the floating-point operand that is compared with.

Operation

This instruction:

1. Compares either:
   - Two floating-point registers.
   - Or one floating-point register and zero.
2. Writes the result to the `FPSCR` flags.

Restrictions

This instruction can optionally raise an `Invalid Operation` exception if either operand is any type of `NaN`. It always raises an `Invalid Operation` exception if either operand is a signaling `NaN`.

Condition flags

When this instruction writes the result to the `FPSCR` flags, the values are normally transferred to the Arm flags by a subsequent `VMRS` instruction.

Example 3-53 Examples

```
VCMP.F32    S4, #0.0VCMP.F32    S4, S2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.7 VCVT and VCVTR between floating-point and integer

Converts a value in a register from floating-point to and from a 32-bit integer.

Syntax

`VCVT{R}{cond}.Tm.F32 Sd, Sm`

`VCVT{cond}.F32.Tm Sd, Sm`

Where:

`R`	If `R` is specified, the operation uses the rounding mode specified by the `FPSCR`. If `R` is omitted. the operation uses the `Round towards Zero` rounding mode.
cond	Is an optional condition code.
Tm	Is the data type for the operand. It must be one of: • `S32` signed 32-bit value. • `U32` unsigned 32-bit value.
Sd, Sm	Are the destination register and the operand register.

Operation

These instructions:

1. Either:
   - Convert a value in a register from floating-point value to a 32-bit integer.
   - Convert from a 32-bit integer to floating-point value.
2. Place the result in a second register.

The floating-point to integer operation normally uses the `Round towards Zero` rounding mode, but can optionally use the rounding mode specified by the `FPSCR`.

The integer to floating-point operation uses the rounding mode specified by the `FPSCR`.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.8 VCVT between floating-point and fixed-point

Converts a value in a register from floating-point to and from fixed-point.

Syntax

`VCVT{cond}.Td.F32 Sd, Sd, #fbits`

`VCVT{cond}.F32.Td Sd, Sd, #fbits`

Where:

cond Is an optional condition code.

Td Is the data type for the fixed-point number. It must be one of:
- `S16` signed 16-bit value.
- `U16` unsigned 16-bit value.
- `S32` signed 32-bit value.
- `U32` unsigned 32-bit value.

Sd Is the destination register and the operand register.

fbits Is the number of fraction bits in the fixed-point number:
- If *Td* is `S16` or `U16`, *fbits* must be in the range 0-16.
- If *Td* is `S32` or `U32`, *fbits* must be in the range 1-32.

Operation

This instruction:

1. Either
   - Converts a value in a register from floating-point to fixed-point.
   - Converts a value in a register from fixed-point to floating-point.
2. Places the result in a second register.

The floating-point values are single-precision or double-precision.

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.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.9 VDIV

Divides floating-point values.

Syntax

```
VDIV{cond}.F32 {Sd,} Sn, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination register.
Sn, Sm	Are the operand registers.

Operation

This instruction:

1. Divides one floating-point value by another floating-point value.
2. Writes the result to the floating-point destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.10 VFMA and VFMS

Floating-point Fused Multiply Accumulate and Subtract.

Syntax

`VFMA{cond}.F32 {Sd,} Sn, Sm`

`VFMS{cond}.F32 {Sd,} Sn, Sm`

Where:

cond	Is an optional condition code.
Sd	Is the destination register.
Sn, Sm	Are the operand registers.

Operation

The `VFMA` instruction:

1. Multiplies the floating-point values in the operand registers.
2. Accumulates the results into the destination register.

The result of the multiply is not rounded before the accumulation.

The `VFMS` instruction:

1. Negates the first operand register.
2. Multiplies the floating-point values of the first and second operand registers.
3. Adds the products to the destination register.
4. Places the results in the destination register.

The result of the multiply is not rounded before the addition.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.11 VFNMA and VFNMS

Floating-point Fused Negate Multiply Accumulate and Subtract.

Syntax

```
VFNMA{cond}.F32 {Sd,} Sn, Sm
```

```
VFNMS{cond}.F32 {Sd,} Sn, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination register.
Sn, Sm	Are the operand registers.

Operation

The `VFNMA` instruction:

1. Negates the first floating-point operand register.
2. Multiplies the first floating-point operand with second floating-point operand.
3. Adds the negation of the floating -point destination register to the product
4. Places the result into the destination register.

The result of the multiply is not rounded before the addition.

The `VFNMS` instruction:

1. Multiplies the first floating-point operand with second floating-point operand.
2. Adds the negation of the floating-point value in the destination register to the product.
3. Places the result in the destination register.

The result of the multiply is not rounded before the addition.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.12 VLDM

Floating-point Load Multiple.

Syntax

`VLDM{mode}{cond}{.size} Rn{!}, list`

Where:

mode	Is the addressing mode:	
	`IA`	Increment after. The consecutive addresses start at the address specified in *Rn*.
	`DB`	Decrement before. The consecutive addresses end before the address specified in *Rn*.
cond	Is an optional condition code.	
size	Is an optional data size specifier.	
Rn	Is the base register. The `SP` can be used.	
`!`	Is the command to the instruction to write a modified value back to *Rn*. This is required if `mode == DB`, and is optional if `mode == IA`.	
list	Is the list of extension registers to be loaded, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.	

Operation

This instruction loads multiple extension registers from consecutive memory locations using an address from an Arm core register as the base address.

Restrictions

The restrictions are:
- If *size* is present, it must be equal to the size in bits, `32` or `64`, of the registers in *list*.
- For the base address, the SP can be used. In the Arm instruction set, if *!* is not specified the `PC` can be used.
- *list* must contain at least one register. If it contains doubleword registers, it must not contain more than 16 registers.
- If using the `Decrement before` addressing mode, the write back flag, *!*, must be appended to the base register specification.

Condition flags

These instructions do not change the flags.

Example 3-54 Examples

```
VLDMIA.F64 r1, {d3,d4,d5}
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.13 VLDR

Loads a single extension register from memory.

Syntax

```
VLDR{cond}{.F<32|64>} <Sd|Dd>, [Rn {, #imm}]
```

```
VLDR{cond}{.F<32|64>} <Sd|Dd>, label
```

```
VLDR{cond}{.F<32|64>} <Sd|Dd>, [PC, #imm]
```

Where:

cond	Is an optional condition code.
32, 64	Are the optional data size specifiers.
Dd	Is the destination register for a doubleword load.
Sd	Is the destination register for a singleword load.
Rn	Is the base register. The SP can be used.
imm	Is the + or - immediate offset used to form the address. Permitted address values are multiples of 4 in the range 0-1020.
label	Is the label of the literal data item to be loaded.

Operation

This instruction loads a single extension register from memory, using a base address from an Arm core register, with an optional offset.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.14 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.

Syntax

`VLLDM {cond}<Rn>`

Where:

cond	Is an optional condition code.
Rn	Is the base register.

Operation

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`.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.15 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.

Syntax

`VLSTM {cond}<Rn>`

Where:

cond	Is an optional condition code.
Rn	Is the base register.

Operation

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`.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.16 VMLA and VMLS

Multiplies two floating-point values, and accumulates or subtracts the result.

Syntax

`VMLA{cond}.F32 Sd, Sn, Sm`

`VMLS{cond}.F32 Sd, Sn, Sm`

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sn, Sm	Are the operand floating-point values.

Operation

The floating-point Multiply Accumulate instruction:

1. Multiplies two floating-point values.
2. Adds the results to the destination floating-point value.

The floating-point Multiply Subtract instruction:

1. Multiplies two floating-point values.
2. Subtracts the products from the destination floating-point value.
3. Places the results in the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.17 VMOV Immediate

Move floating-point Immediate.

Syntax

`VMOV{cond}.F32 Sd, #imm`

Where:

cond	Is an optional condition code.
Sd	Is the destination register.
imm	Is a floating-point constant.

Operation

This instruction copies a constant value to a floating-point register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.18 VMOV Register

Copies the contents of one register to another.

Syntax

```
VMOV{cond}.F<32> Sd, Sm Dm
```

Where:

cond	Is an optional condition code.
Dd	Is the destination register, for a doubleword operation.
Dm	Is the source register, for a doubleword operation.
Sd	Is the destination register, for a singleword operation.
Sm	Is the source register, for a singleword operation.

Operation

This instruction copies the contents of one floating-point register to another.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.19 VMOV scalar to core register

Transfers one word of a doubleword floating-point register to an Arm core register.

Syntax

`VMOV{cond} Rt, Dn[x]`

Where:

cond	Is an optional condition code.
Rt	Is the destination Arm core register.
Dn	Is the 64-bit doubleword register.
x	Specifies which half of the doubleword register to use: • If *x* is 0, use lower half of doubleword register. • If *x* is 1, use upper half of doubleword register.

Operation

This instruction transfers one word from the upper or lower half of a doubleword floating-point register to an Arm core register.

Restrictions

Rt cannot be PC or SP.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.20 VMOV core register to single-precision

Transfers a single-precision register to and from an Arm core register.

Syntax

`VMOV{cond} Sn, Rt`

`VMOV{cond} Rt, Sn`

Where:

cond	Is an optional condition code.
<Sn>	Is the single-precision floating-point register.
Rt	Is the Arm core register.

Operation

This instruction transfers:

- The contents of a single-precision register to an Arm core register.
- The contents of an Arm core register to a single-precision register.

Restrictions

Rt cannot be PC or SP.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.21 VMOV two core registers to two single-precision registers

Transfers two consecutively numbered single-precision registers to and from two Arm core registers.

Syntax

`VMOV{cond} Sm, Sm1, Rt, Rt2`

`VMOV{cond} Rt, Rt2, Sm, Sm1`

Where:

cond	Is an optional condition code.
Sm	Is the first single-precision register.
Sm1	Is the second single-precision register. This is the next single-precision register after *Sm*.
Rt	Is the Arm core register that *Sm* is transferred to or from.
Rt2	Is the Arm core register that *Sm1* is transferred to or from.

Operation

This instruction transfers:

- The contents of two consecutively numbered single-precision registers to two Arm core registers.
- The contents of two Arm core registers to a pair of single-precision registers.

Restrictions

The restrictions are:

- The floating-point registers must be contiguous, one after the other.
- The Arm core registers do not have to be contiguous.
- *Rt* cannot be PC or SP.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.22 VMOV two core registers and a double-precision register

Transfers two words from two Arm core registers to a doubleword register, or from a doubleword register to two Arm core registers.

Syntax

`VMOV{cond} Dm, Rt, Rt2`

`VMOV{cond} Rt, Rt2, Dm`

Where:

cond	Is an optional condition code.
Dm	Is the double-precision register.
Rt, Rt2	Are the two Arm core registers.

Operation

This instruction:

- Transfers two words from two Arm core registers to a doubleword register.
- Transfers a doubleword register to two Arm core registers.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.23 VMOV core register to scalar

Transfers one word to a floating-point register from an Arm core register.

Syntax

```
VMOV{cond}{.32} Dd[x], Rt
```

Where:

cond	Is an optional condition code.
32	Is an optional data size specifier.
Dd[*x*]	Is the destination, where [*x*] defines which half of the doubleword is transferred, as follows: • If *x* is 0, the lower half is extracted. • If *x* is 1, the upper half is extracted.
Rt	Is the source Arm core register.

Operation

This instruction transfers one word to the upper or lower half of a doubleword floating-point register from an Arm core register.

Restrictions

Rt cannot be PC or SP.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.24 VMRS

Move to Arm Core register from floating-point System Register.

Syntax

`VMRS{cond} Rt, FPSCR`

`VMRS{cond} APSR_nzcv, FPSCR`

Where:

cond	Is an optional condition code.
Rt	Is the destination Arm core register. This register can be R0-R14.
APSR_nzcv	Transfer floating-point flags to the APSR flags.

Operation

This instruction performs one of the following actions:
- Copies the value of the `FPSCR` to a general-purpose register.
- Copies the value of the `FPSCR` flag bits to the APSR N, Z, C, and V flags.

Restrictions

Rt cannot be PC or SP.

Condition flags

These instructions optionally change the N, Z, C, and V flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.25 VMSR

Move to floating-point System Register from Arm Core register.

Syntax

`VMSR{cond} FPSCR, Rt`

Where:

cond	Is an optional condition code.
Rt	Is the general-purpose register to be transferred to the FPSCR.

Operation

This instruction moves the value of a general-purpose register to the FPSCR.

Restrictions

Rt cannot be PC or SP.

Condition flags

This instruction updates the FPSCR.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.26 VMUL

Floating-point Multiply.

Syntax

```
VMUL{cond}.F32 {Sd,} Sn, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sn, Sm	Are the operand floating-point values.

Operation

This instruction:

1. Multiplies two floating-point values.
2. Places the results in the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.27 VNEG

Floating-point Negate.

Syntax

```
VNEG{cond}.F32 Sd, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sm	Is the operand floating-point value.

Operation

This instruction:

1. Negates a floating-point value.
2. Places the results in a second floating-point register.

The floating-point instruction inverts the sign bit.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.28 VNMLA, VNMLS and VNMUL

Floating-point multiply with negation followed by add or subtract.

Syntax

`VNMLA{cond}.F32 Sd, Sn, Sm`

`VNMLS{cond}.F32 Sd, Sn, Sm`

`VNMUL{cond}.F32 {Sd,} Sn, Sm`

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point register.
Sn, Sm	Are the operand floating-point registers.

Operation

The `VNMLA` instruction:

1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the negation of the product.
3. Writes the result back to the destination register.

The `VNMLS` instruction:

1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the product.
3. Writes the result back to the destination register.

The `VNMUL` instruction:

1. Multiplies together two floating-point register values.
2. Writes the negation of the result to the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.29 VPOP

Floating-point extension register Pop.

Syntax

`VPOP{cond}{.size} list`

Where:

cond	Is an optional condition code.
size	Is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.
list	Is a list of extension registers to be loaded, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

Operation

This instruction loads multiple consecutive extension registers from the stack.

Restrictions

list must contain at least one register, and not more than sixteen registers.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.30 VPUSH

Floating-point extension register Push.

Syntax

`VPUSH{cond}{.size} list`

Where:

cond	Is an optional condition code.
size	Is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.
list	Is a list of the extension registers to be stored, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

Operation

This instruction stores multiple consecutive extension registers to the stack.

Restrictions

list must contain at least one register, and not more than sixteen.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.31 VSQRT

Floating-point Square Root.

Syntax

```
VSQRT{cond}.F32 Sd, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sm	Is the operand floating-point value.

Operation

This instruction:

- Calculates the square root of the value in a floating-point register.
- Writes the result to another floating-point register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.32 VSTM

Floating-point Store Multiple.

Syntax

`VSTM{mode}{cond}{.size} Rn{!}, list`

Where:

mode	Is the addressing mode: • IA *Increment After*. The consecutive addresses start at the address specified in *Rn*. This is the default and can be omitted. • DB *Decrement Before*. The consecutive addresses end just before the address specified in *Rn*.
cond	Is an optional condition code.
size	Is an optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.
Rn	Is the base register. The SP can be used.
!	Is the function that causes the instruction to write a modified value back to *Rn*. Required if `mode == DB`.
list	Is a list of the extension registers to be stored, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

Operation

This instruction stores multiple extension registers to consecutive memory locations using a base address from an Arm core register.

Restrictions

The restrictions are:
- *list* must contain at least one register. If it contains doubleword registers it must not contain more than 16 registers.
- Use of the PC as *Rn* is deprecated.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.33 VSTR

Floating-point Store.

Syntax

```
VSTR{cond}{.32} Sd, [Rn{, #imm}]
```

```
VSTR{cond}{.64} Dd, [Rn{, #imm}]
```

Where:

cond	Is an optional condition code.
32, 64	Are the optional data size specifiers.
Sd	Is the source register for a singleword store.
Dd	Is the source register for a doubleword store.
Rn	Is the base register. The SP can be used.
imm	Is the + or - immediate offset used to form the address. Values are multiples of 4 in the range 0-1020. *imm* can be omitted, meaning an offset of `+0`.

Operation

This instruction stores a single extension register to memory, using an address from an Arm core register, with an optional offset, defined in *imm*:

Restrictions

The use of PC for *Rn* is deprecated.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.34 VSUB

Floating-point Subtract.

Syntax

```
VSUB{cond}.F32 {Sd,} Sn, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sn, Sm	Are the operand floating-point values.

Operation

This instruction:

1. Subtracts one floating-point value from another floating-point value.
2. Places the results in the destination floating-point register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.35 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.

Syntax

`VSEL{cond}.F32 Sd, Sn, Sm`

Where:

cond	Is an optional condition code. `VSEL` has a subset of the condition codes. The condition codes for `VSEL` are limited to `GE`, `GT`, `EQ` and `VS`, with the effect that `LT`, `LE`, `NE` and `VC` is achievable by exchanging the source operands.
Sd	Is the destination single-precision floating-point value.
Sn, Sm	Are the operand single-precision floating-point values.

Operation

Depending on the result of the condition code, this instruction moves either:

- *Sn* source register to the destination register.
- *Sm* source register to the destination register.

The behavior is:

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
S[d] = if ConditionHolds(cond) then S[n] else S[m];
```

Restrictions

The `VSEL` instruction must not occur inside an IT block.

Condition flags

This instruction does not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.36 VCVTA, VCVTM VCVTN, and VCVTP

Floating-point to integer conversion with directed rounding.

Syntax

`VCVT<rmode>.S32.F32 Sd, Sm`

`VCVT<rmode>.U32.F32 Sd, Sm`

Where:

Sd	Is the destination single-precision or double-precision floating-point value.
Sm,	Are the operand single-precision or double-precision floating-point values.
<rmode>	Is one of:

`A`	Round to nearest ties away.
`M`	Round to nearest even.
`N`	Round towards plus infinity.
`P`	Round towards minus infinity.

Operation

These instructions:

1. Read the source register.
2. Convert to integer with directed rounding.
3. Write to the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.37 VCVTB and VCVTT

Converts between half-precision and single-precision without intermediate rounding.

Syntax

`VCVT{y}{cond}.F32.F16 Sd, Sm`

`VCVT{y}{cond}.F16.F32 Sd, Sm`

Where:

y	Specifies which half of the operand register *Sm* or destination register *Sd* is used for the operand or destination: • If *y* is `B`, then the bottom half, bits [15:0], of *Sm* or *Sd* is used. • If *y* is `T`, then the top half, bits [31:16], of *Sm* or *Sd* is used.
cond	Is an optional condition code.
Sd	Is the destination register.
Sm	Is the operand register.

Operation

This instruction with the `.F16.F32` suffix:

1. Converts the half-precision value in the top or bottom half of a single-precision register to single-precision value.
2. Writes the result to a single-precision register.

This instruction with the `.F32.F16` suffix:

1. Converts the value in a single-precision register to half-precision value.
2. Writes the result into the top or bottom half of a single-precision register, preserving the other half of the target register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.38 VMAXNM and VMINNM

Return the minimum or the maximum of two floating-point numbers with NaN handling as specified by IEEE754-2008.

Syntax

`VMAXNM.F32 Sd, Sn, Sm`

`VMINNM.F32 Sd, Sn, Sm`

Where:

Sd	Is the destination single-precision floating-point value.
Sn, Sm	Are the operand single-precision floating-point values.

Operation

The `VMAXNM` instruction compares two source registers, and moves the largest to the destination register.

The `VMINNM` instruction compares two source registers, and moves the smallest to the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.39 VRINTR and VRINTX

Round a floating-point value to an integer in floating-point format.

Syntax

```
VRINT{R,X}{cond}.F32 Sd, Sm
```

Where:

cond	Is an optional condition code.
Sd	Is the destination floating-point value.
Sm	Are the operand floating-point values.

Operation

These instructions:

1. Read the source register.
2. Round to the nearest integer value in floating-point format using the rounding mode specified by 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.
3. Write the result to the destination register.
4. For the `VRINTX` instruction only. Generate a floating-point exception if the result is not numerically equal to the input value.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.20.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ

Round a floating-point value to an integer in floating-point format using directed rounding.

Syntax

`VRINT<rmode>.F32 Sd, Sm`

Where:

Sd	Is the destination single-precision floating-point value.
Sm	Are the operand single-precision floating-point values.
<rmode>	Is one of:

`A`	Round to nearest ties away.
`N`	Round to Nearest Even.
`P`	Round towards Plus Infinity.
`M`	Round towards Minus Infinity.
`Z`	Round towards Zero.

Operation

These instructions:

1. Read the source register.
2. Round to the nearest integer value with a directed rounding mode specified by the instruction.
3. 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.
4. Write the result to the destination register.

Restrictions

`VRINTA`, `VRINTN`, `VRINTP` and `VRINTM` cannot be conditional. `VRINTZ` can be conditional.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.21 Arm®v8.1-M shift, saturate, and reverse operations instructions

Reference material for the Cortex-M55 processor Armv8.1-M shift, saturate, and reverse operations instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.21.1 List of Arm®v8.1-M shift, saturate, and reverse operations instructions

An alphabetically ordered list of the Armv8.1-M shift, saturate, and reverse operations instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-15 Armv8.1-M shift, saturate, and reverse operations instructions

Mnemonic	Brief description	See
`ASRL`	`ASRL` (immediate) Arithmetic Shift Right Long	*3.21.2 ASRL (immediate)* on page 3-262
`ASRL`	`ASRL` (register) Arithmetic Shift Right Long	*3.21.3 ASRL (register)* on page 3-263
`LSLL`	`LSLL` (immediate) Logical Shift Left Long	*3.21.4 LSLL (immediate)* on page 3-263
`LSLL`	`LSLL` (register) Logical Shift Left Long	*3.7.2 LSLL* on page 3-115
`LSRL`	`LSRL` (immediate) Logical Shift Right Long	*3.21.4 LSLL (immediate)* on page 3-263
`SQRSHR`	`SQRSHR` (register) Signed Saturating Rounding Shift Right	*3.6.7 SQRSHR* on page 3-113
`SQRSHRL`	`SQRSHRL` (register) Signed Saturating Rounding Shift Right Long	*3.6.8 SQRSHRL* on page 3-114
`SQSHL`	`SQSHL` (immediate) Signed Saturating Shift Left	*3.21.9 SQSHL (immediate)* on page 3-267
`SQSHLL`	`SQSHLL` (immediate) Signed Saturating Shift Left Long	*3.21.10 SQSHLL (immediate)* on page 3-268
`SRSHR`	`SRSHR` (immediate) Signed Rounding Shift Right	*3.21.11 SRSHR (immediate)* on page 3-268
`SRSHRL`	`SRSHRL` (immediate) Signed Rounding Shift Right Long	*3.21.12 SRSHRL (immediate)* on page 3-269
`UQRSHL`	`UQRSHL` (register) Unsigned Saturating Rounding Shift Left	*3.7.7 UQRSHL* on page 3-118
`UQRSHLL`	`UQRSHLL` (register) Unsigned Saturating Rounding Shift Left Long	*3.21.14 UQRSHLL (register)* on page 3-270
`UQSHL`	`UQSHL` (immediate) Unsigned Saturating Shift Left	*3.21.15 UQSHL (immediate)* on page 3-271
`UQSHLL`	`UQSHLL` (immediate) Unsigned Saturating Shift Left Long	*3.21.16 UQSHLL (immediate)* on page 3-272
`URSHR`	`URSHR` (immediate) Unsigned Rounding Shift Right	*3.21.17 URSHR (immediate)* on page 3-273
`URSHRL`	`URSHRL` (immediate) Unsigned Rounding Shift Right Long	*3.21.18 URSHRL (immediate)* on page 3-273
`VBRSR`	Vector Bit Reverse and Shift Right	*3.21.19 VBRSR* on page 3-274
`VMOVL`	Vector Move Long	*3.21.20 VMOVL* on page 3-275
`VMOVN`	Vector Move and Narrow	*3.21.21 VMOVN* on page 3-275
`VQMOVN`	Vector Saturating Move and Narrow	*3.21.22 VQMOVN* on page 3-276
`VQMOVUN`	Vector Saturating Move Unsigned and Narrow	*3.21.23 VQMOVUN* on page 3-277

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-15 Armv8.1-M shift, saturate, and reverse operations instructions (continued)

Mnemonic	Brief description	See
`VQRSHL`	Vector Saturating Rounding Shift Left	*3.21.24 VQRSHL* on page 3-278
`VQRSHRN`	Vector Saturating Rounding Shift Right and Narrow	*3.21.25 VQRSHRN* on page 3-279
`VQRSHRUN`	Vector Saturating Rounding Shift Right Unsigned and Narrow	*3.21.26 VQRSHRUN* on page 3-280
`VQSHL, VQSHLU`	Vector Saturating Shift Left, Vector Saturating Shift Left Unsigned	*3.21.27 VQSHL, VQSHLU* on page 3-281
`VQSHRN`	Vector Saturating Shift Right and Narrow	*3.21.28 VQSHRN* on page 3-282
`VQSHRUN`	Vector Saturating Shift Right Unsigned and Narrow	*3.21.29 VQSHRUN* on page 3-283
`VRSHL`	Vector Rounding Shift Left	*3.21.30 VRSHL* on page 3-284
`VRSHR`	Vector Rounding Shift Right	*3.21.31 VRSHR* on page 3-285
`VRSHRN`	Vector Rounding Shift Right and Narrow	*3.21.32 VRSHRN* on page 3-286
`VSHL`	Vector Shift Left	*3.21.33 VSHL* on page 3-287
`VSHLC`	Whole Vector Left Shift with Carry	*3.21.34 VSHLC* on page 3-288
`VSHLL`	Vector Shift Left Long	*3.21.35 VSHLL* on page 3-289
`VSHR`	Vector Shift Right	*3.21.36 VSHR* on page 3-290
`VSHRN`	Vector Shift RightVector Shift Right and Narrow	*3.21.37 VSHRN* on page 3-291
`VSLI`	Vector Shift Left and Insert	*3.21.38 VSLI* on page 3-292
`VSRI`	Vector Shift Right and Insert	*3.21.39 VSRI* on page 3-293

3.21.2 ASRL (immediate)

Arithmetic Shift Right Long.

Syntax

```
ASRL<c> RdaLo, RdaHi, #<imm>
```

Parameters

`RdaHi` General-purpose register for the high-half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

`RdaLo` General-purpose register for the low-half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

`c` See Standard Assembler Syntax Fields

`imm` The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

Operation

Arithmetic shift right by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

ASRL (immediate) example

```
// Scaled Dot-product, right shift long by immediate 6
VLDRW.U32       Q0, [R2]        // Contigous load Q1, base address in R2
                                // R2 points to memory containing {TOQ31(0.1) TOQ31(0.2)
                                // TOQ31(0.3) TOQ31(0.4)}
                                // Q31(x) indicates fixed-point format
                                // = {214748368 429496736 644245120 858993472}
VRMLALVH.s32    R0, R1, Q0, Q0  // 32-bit sum of square
                                // R0:R1 = (sum(Q0[i] * Q0[i]) + (1<<7)) >> 8    i={0..3}
                                // R0:R1 = 5404319794436509 (= 0.3 * 2^54)
                                // Accumulator in Q10.54
                                // Q10.54 indicates fixed-point format
ASRL            R0, R1, #6      // R0:R1 = R0:R1 >> 6
                                // convert in Q10.54 to Q16.48 fixed-point format
                                // R0:R1 = 84442496788070 (= 0.3 * 2^48)
```

3.21.3 ASRL (register)

Arithmetic Shift Right Long.

Syntax

```
ASRL<c> RdaLo, RdaHi, Rm
```

Parameters

`RdaHi` General-purpose register for the high-half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

`RdaLo` General-purpose register for the low-half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

`Rm` General-purpose source register holding a shift amount in its bottom 8 bits.

`c` See Standard Assembler Syntax Fields

Restrictions

- `Rm` must not use the same register as `RdaLo`.
- `Rm` must not use the same register as `SP` and `PC`.
- `Rm` must not use the same register as `RdaHi`.
- `RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

Operation

Arithmetic shift right by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

ASRL (register) example

```
MOVW         R2, #0
MOVT         R2, 0x7000        // R2 = 0x70000000
MOV          R3, #1            // Right shift value = 1
SMULL        R0, R1, R2, R2    // R0:R1 = R2^2 = 0x3100000000000000
ASRL         R0, R1, R3        // R0:R1 = R0:R1 >> 1 = 0x1880000000000000

// Arithmetic left shift (negative right shift)
MOVW         R2, #0
MOVT         R2, 0x7000        // R2 = 0x70000000
MOV          R4, #-1           // Right shift value = -1 (left)
SMULL        R0, R1, R2, r2    // R0:R1 = R2^2 = 0x310000000000000
ASRL         R0, R1, R4        // R0:R1 = R0:R1 << 1 = 0x6200000000000000
```

3.21.4 LSLL (immediate)

Logical Shift Left Long.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
LSLL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher-half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower-half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`

Post-conditions

There are no condition flags.

Operation

Logical shift left by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

LSLL (immediate) example

```
// Logical long shift left by 8 (immediate)
MOVW         R2, #0x3311
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877   // R2:R3 = 0x8877665544332211

LSLL         R2, R3, #8  // R2:R3 = R2:R3 << 8 = 0x7766554433331100
                         //(The Most Significant Byte is removed)
```

3.21.5 LSLL (register)

Logical Shift Left Long.

Syntax

```
LSLL<c> RdaLo, RdaHi, Rm
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

Rm General-purpose source register holding a shift amount in its bottom 8 bits.

c See Standard Assembler Syntax Fields

Restrictions

- `Rm` must not use the same register as `RdaLo`.
- `Rm` must not use the same register as `SP` and `PC`.
- `Rm` must not use the same register as `RdaHi`.
- `RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Logical shift left by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

LSLL (register) example

```
// Logical long left shift
MOVW         R2, #0xFFFF
MOVT         R2, #0x7FFF        // R2 = 0x7FFFFFFF
MOV          R3, #2             // Left shift value = 2
SMULL        R0, R1, R2, R2     // R0:R1 = R2^2 = 0x3FFFFFFF00000001
LSLL         R0, R1, R3         // R0:R1 = R0:R1 << 2 = 0xFFFFFFFC00000004
                                // (The two Most Significant Bits are removed)

// Logical long right shift (negative shift value)
MOVW         R2, #0xFFFF
MOVT         R2, #0xFFFF        // R2 = 0xFFFFFFFF
MOVW         R3, #0xFFFF
MOVT         R3, #0x7FFF        // R3 = 0x7FFFFFFF
MOV          R4, #-2            // Left shift value = -2 (right)
SMULL        R0, R1, R2, R3     // R0:R1 = R2*R3 = 0xFFFFFFFF80000001
LSLL         R0, R1, R4         // R0:R1 = R0:R1 >> 2 = 0x3FFFFFFFE0000000
                                //(The two Least Significant Bits are removed)
```

3.21.6 LSRL (immediate)

Logical Shift Right Long.

Syntax

```
LSRL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

Operation

Logical shift right by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

LSRL (immediate) example

```
// Logical long shift right by 8 (immediate)
MOVW         R2, #0x2211
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877       // R2:R3 = 0x8877665544332211

LSRL         R2, R3, #8        // R3:R3 = R3:R3 >> 8 = 0x0088776655443322
                               //(The Least Significant Byte is removed)
```

3.21.7 SQRSHR (register)

Signed Saturating Rounding Shift Right.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
SQRSHR<c> Rda, Rm
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

Rm General-purpose source register holding a shift amount in its bottom 8 bits.

c See Standard Assembler Syntax Fields

Restrictions

- `Rm` must not use the same register as `SP` and `PC`.
- `Rm` must not use the same register as `Rda`.
- `Rda` must not use the same register as `SP` and `PC`.

Post-conditions

Updates Q flag in APSR register.

Operation

Signed saturating rounding shift right by 0 to 32 bits of a 32-bit value stored in a general-purpose register. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

SQRSHR (register) example

```
// Signed Saturating Rounding Shift Right by 4
MOV          R0, #4          // Right shift amount
MOVW         R2, #0x2218
MOVT         R2, #0x4433     // R2 = 0x44332218
SQRSHR       R2, R0          // R2 = (R2 + (1 << (4-1))) >> 4 = 0x04433222


 // Signed Saturating Rounding Shift left by 4 (negative right shift)
MOV          R1, #-4         // Right shift amount
                             //(Treated as a left shift because amount is negative)
MOVW         R2, #0x2218

MOVT         R2, #0x4433     // R2 = 0x44332218

SQRSHR       R2, R1          // R2 = SSAT32(R2 << 4)= SSAT32(0x443322180) = 0x7FFFFFFF
```

3.21.8 SQRSHRL (register)

Signed Saturating Rounding Shift Right Long.

Syntax

```
SQRSHRL<c> RdaLo, RdaHi, #<saturate>, Rm
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

Rm General-purpose source register holding a shift amount in its bottom 8 bits.

c See Standard Assembler Syntax Fields

saturate The bit position for saturation. This parameter must be one of the following values:

- `#64`.
- `#48`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

- `Rm` must not use the same register as `RdaLo`.
- `Rm` must not use the same register as `SP` and `PC`.
- `Rm` must not use the same register as `RdaHi`.
- `RdaHi` must not use `SP`.

Post-conditions

Updates Q flag in APSR register.

Operation

Signed saturating rounding shift right by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

SQRSHRL (register) example

```
// Signed Saturating Rounding Shift Right by 4 Long
MOV          R0, #4          // Right shift amount = 4
MOVW         R2, #0x2218
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x0000      // R2:R3 = 0x0000665544332218
SQRSHRL      R2, R3, #48, R0  // R2:R3 = (R2:R3 + (1 << (4-1))) >> 4) = 0x0000066554433222

// Signed 48-bit Saturating Rounding Shift left by 4 Long (negative right shift)
MOV          R0, #-4          // Right shift amount = -4 (left shift)
MOVW         R2, #0x3318
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x0000      // R2:R3 = 0x0000665544332218
SQRSHRL      R2, R3, #48, R0  // R2:R3 = SSAT48(R2:R3 << 4) = 0x00007FFFFFFFFFFF

// Signed 64-bit Saturating Rounding Shift left by 4 Long (negative right shift)
MOV          R0, #-4          // Right shift amount = -4 (left shift)
MOVW         R2, #0x3318
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x0000      // R2:R3 = 0x0000665544332218
SQRSHRL      R2, R3, #64, R0  // R2:R3 = SSAT64(R2:R3 << 4) 0x0006655443322180
```

3.21.9 SQSHL (immediate)

Signed Saturating Shift Left.

Syntax

```
SQSHL<c> Rda, #<imm>
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`Rda` must not use the same register as `SP` and `PC`.

Post-conditions

Updates Q flag in APSR register.

Operation

Signed saturating shift left by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

SQSHL (immediate) example

```
// Signed Saturating Shift Left by immediate 2 (no saturation)
MOVW         R2, #0x2218
MOVT         R2, #0x0433   // R2 = 0x04332218
SQSHL        R2, #2        // R2 = SSAT32(R2 << 2) = 0x10CC8860

// Signed Saturating Shift Left by immediate 5 (saturation)
MOVW         R2, #0x2218
MOVT         R2, #0x0433   // R2 = 0x04332218
SQSHL        R2, #5        // R2 = SSAT32(R2 << 5) = 0x7FFFFFFF
```

3.21.10 SQSHLL (immediate)

Signed Saturating Shift Left Long.

Syntax

```
SQSHLL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`.

Post-conditions

Updates Q flag in APSR register.

Operation

Signed saturating shift left by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

SQSHLL (immediate) example

```
// Signed Saturating Shift Left Long by immediate 2 (no saturation)
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0        // R2:R3 = 0x0433221800000000

SQSHLL       R2, R3, #2    // R2:R3 = SSAT64(R2:R3 << 2) = 0x10CC886000000000


// Signed Saturating Shift Left Long by immediate 5 (saturation)
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0        // R2:R3 = 0x0433221800000000
SQSHLL       R2, R3, #5    // R2:R3 = SSAT64(R2:R3 << 5) = 0x7FFFFFFFFFFFFFFF (saturation)
```

3.21.11 SRSHR (immediate)

Signed Rounding Shift Right.

Syntax

```
SRSHR<c> Rda, #<imm>
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`Rda` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

Operation

Signed rounding shift right by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

SRSHR (immediate) example

```
// Signed Rounding Shift Right by immediate 4
MOVW         R2, #0x2218
MOVT         R2, #0x0433   // R2 = 0x04332218

SRSHR        R2, #4       // R2 = (R2 + (1 << (4-1))) >> 4) = 0x00433222
```

3.21.12 SRSHRL (immediate)

Signed Rounding Shift Right Long.

Syntax

```
SRSHRL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`

Post-conditions

There are no condition flags.

Operation

Signed rounding shift right by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

SRSHRL (immediate) example

```
// Signed Rounding Shift Right Long by immediate 4
MOVW         R2, #0x2218
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877        // R2:R3 = 0x8877665544332218

SRSHRL       R2, R3, #4        // R2:R3 = (R2:R3 + (1 << (4-1))) >> 4) = 0xF887766554433222
```

3.21.13 UQRSHL (register)

Unsigned Saturating Rounding Shift Left.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
UQRSHL<c> Rda, Rm
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

Rm General-purpose source register holding a shift amount in its bottom 8 bits.

c See Standard Assembler Syntax Fields

Restrictions

- `Rm` must not use the same register as `SP` and `PC`.
- `Rm` must not use the same register as `Rda`.
- `Rda` must not use the same register as `SP` and `PC`.

Post-conditions

Updates Q flag in APSR register.

Operation

Unsigned saturating rounding shift left by 0 to 32 bits of a 32-bit value stored in a general-purpose register. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

UQRSHL (register) example

```
// Unsigned Saturating Rounding Shift Left by 5
MOV          R0, #5       // Left shift amount
MOVW         R2, #0x2218
MOVT         R2, #0x0433  // R2 = 0x04332218
UQRSHL       R2, R0       // R2 = USAT32(R2 << 5) = USAT32(0x04332218) = 0x86644300


// Unsigned Saturating Rounding Shift right by 5 (negative right shift)
MOV          R1, #-4       // Left shift amount (negative, so right shift)
MOVW         R2, #0x2218
MOVT         R2, #0x0433   // R2 = 0x04332218
UQRSHL       R2, R1        // R2 = (R2 + (1 << (4-1))) >> 4) = (R2 + 8) >> 4 = 0x00433222


// Unsigned Saturating Rounding Shift Left by 6 (saturation)
MOV          R0, #6        // Left shift amount
MOVW         R2, #0x2218
MOVT         R2, #0x0433   // R2 = 0x04332218
UQRSHL       R2, R0        // R2 = USAT32(R2 << 6) = USAT32(0x04332218) = 0xFFFFFFFF
(Saturation)
```

3.21.14 UQRSHLL (register)

Unsigned Saturating Rounding Shift Left Long.

Syntax

```
UQRSHLL<c> RdaLo, RdaHi, #<saturate>, Rm
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

Rm General-purpose source register holding a shift amount in its bottom 8 bits.

c See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`saturate` The bit position for saturation. This parameter must be one of the following values:

- `#64`.
- `#48`.

Restrictions

- `Rm` must not use the same register as `RdaLo`
- `Rm` must not use the same register as `SP` and `PC`
- `Rm` must not use the same register as `RdaHi`
- `RdaHi` must not use `SP`

Post-conditions

Updates Q flag in APSR register.

Operation

Unsigned saturating rounding shift left by 0 to 64 bits of a 64-bit value stored in two general-purpose registers. The shift amount is read in as the bottom byte of Rm. If the shift amount is negative, the shift direction is reversed.

UQRSHLL (register) example

```
// Unsigned 64-bit Saturating Rounding Shift Left Long by 5.
MOV          R0, #5            //  Left shift amount
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0            // R2:R3 = 0x0433221800000000
UQRSHLL      R2, R3, #64, R0   // R2:R3 = USAT64(R2:R3 << 5) = 0x8664430000000000

// Unsigned 64-bit Saturating Rounding Shift right Long by 4 (negative shift)
MOV          R1, #-4         //  left shift amount (negative, so right shift)
MOVW         R2, #0x2218
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877       // R2:R3 = 0x8877665544332218
UQRSHLL      R2, R3, #64, R1  // R2:R3 = (R2:R3 + (1 << (4-1))) >> 4) = (R2:R3 + 8) >> 4 =
0x0887766554433222


// Unsigned 64-bit Saturating Rounding Shift Left Long by 6 (saturation)
MOV          R0, #6
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0           // R2:R3 = 0x0433221800000000
UQRSHLL      R2, R3, #64, r0  // R2:R3 = USAT64(R2:R3 << 6) = 0xFFFFFFFFFFFFFFFF (64-bit
saturation)

// Unsigned 48-bit Saturating Rounding Shift Left Long by 4 (saturation)
MOV          R1, #4
MOV          R3, #0x1000
MOV          R2, #0           // R2:R3 = 0x0000100000000000
UQRSHLL      R2, R3, #48, R1  // R2:R3 = USA48(R2:R3 << 4) = 0x0000FFFFFFFFFFFF   (48-bit
saturation)
```

3.21.15 UQSHL (immediate)

Unsigned Saturating Shift Left.

Syntax

```
UQSHL<c> Rda, #<imm>
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

Rda must not use the same register as SP and PC.

Post-conditions

Updates Q flag in APSR register.

Operation

Unsigned saturating shift left by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

UQSHL (immediate) example

```
// Unsigned Saturating Shift Left by immediate 5
MOVW        R2, #0x2218
MOVT        R2, #0x0433   // R2 = 0x04332218
UQSHL       R2, #5        // R2 = USAT32(r2 << 5) = 0x86644300


// Unsigned Saturating Shift Left by immediate 6 (saturation)
MOVW        R2, #0x2218
MOVT        R2, #0x0433   // R2 = 0x04332218
UQSHL       R2, #6        // R2 = USAT32(R2 << 6) = 0xFFFFFFFF (saturation)
```

3.21.16 UQSHLL (immediate)

Unsigned Saturating Shift Left Long.

Syntax

```
UQSHLL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

RdaHi must not use SP.

Post-conditions

Updates Q flag in APSR register.

Operation

Unsigned saturating shift left by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

UQSHLL (immediate) example

```
// Unsigned Saturating Shift Left Long by immediate 5
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0         // R2:R3 = 0x0433221800000000
UQSHLL       R2, R3, #5     // R2:R3 = USAT64(R2:R3 << 5) = 0x8664430000000000


// Unsigned Saturating Shift Left Long by immediate 6 (saturation)
MOVW         R3, #0x2218
MOVT         R3, #0x0433
MOV          R2, #0         // R2:R3 = 0x0433221800000000
UQSHLL       R2, R3, #6     // R2:R3 = USAT64(R2:R3 << 6) = 0xFFFFFFFFFFFFFFFF
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.21.17 URSHR (immediate)

Unsigned Rounding Shift Right.

Syntax

```
URSHR<c> Rda, #<imm>
```

Parameters

Rda General-purpose source and destination register, containing the value to be shifted.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`Rda` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

Operation

Unsigned rounding shift right by 1 to 32 bits of a 32-bit value stored in a general-purpose register.

URSHR (immediate) example

```
// Unsigned Rounding Shift Right by immediate 4
MOVW         R2, #0x8888
MOVT         R2, #0xF433    // R2 = 0xF4338888
URSHR        R2, #4        // R2 = (R2 + (1 << (4-1))) >> 4)  = 0x0F433889

// Unsigned Rounding Shift Right by immediate 16
MOVW         R2, #0x8888
MOVT         R2, #0xF433   // R2 = 0xF4338888
URSHR        R2, #16       // R2 = (R2 + (1 << (16-1))) >> 16) = 0x0000F434
```

3.21.18 URSHRL (immediate)

Unsigned Rounding Shift Right Long.

Syntax

```
URSHRL<c> RdaLo, RdaHi, #<imm>
```

Parameters

RdaHi General-purpose register for the higher half of the 64-bit source and destination, containing the value to be shifted. This must be an odd numbered register.

RdaLo General-purpose register for the lower half of the 64-bit source and destination, containing the value to be shifted. This must be an even numbered register.

c See Standard Assembler Syntax Fields

imm The number of bits to shift by, in the range 1-32.

Restrictions

`RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Unsigned rounding shift right by 1 to 32 bits of a 64-bit value stored in two general-purpose registers.

URSHRL (immediate) example

```
// Unsigned Rounding Shift Right Long by immediate 4
MOVW         R2, #0x8888
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877     // R2:R3 = 0x8877665544338888

URSHRL       R2, R3, #4     // R2:R3 = (R2:R3 + (1 << (4-1))) >> 4) = (R2:R3 + 8) >> 4
                            // = 0x0887766554433889


// Unsigned Rounding Shift Right Long by immediate 16
MOVW         R2, #0x8888
MOVT         R2, #0x4433
MOVW         R3, #0x6655
MOVT         R3, #0x8877     // R2:R3 = 0x8877665544338888
URSHRL       R2, R3, #16     // R2:R3 = (R2:R3 + (1 << (16-1))) >> 16) = (R2:R3 + 0x8000) >>
16
                             // = 0x0000887766554434
```

3.21.19 VBRSR

Vector Bit Reverse and Shift Right.

Syntax

```
VBRSR<v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qn Source vector register.

Rm General-purpose register containing the number of least significant bits to reverse in its bottom 8 bits.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - 8
  - 16
  - 32

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

Operation

Reverse the specified number of least significant bits in each element of a vector register and set the other bits to zero. The number of bits to reverse is read in from the bottom byte of `Rm` and clamped to the range 0-(`dt`-1).

VBRSR example

```
// Bit Reversal of an incrementing 8-bit vector
MOV           R2, #0
VIDUP.U8      Q0, R2, #1   // Generates 8-bit incrementing sequence
MOV           R0, #4       // Set bit-reverse point to 16, 4 = log2(16)
                           // Q0 = [ 0x0 0x1 0x2 0x3  0x4 0x5 0x6 0x7
                           // 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF ]
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VBRSR.8       Q1, Q0, R0   // Bit-reverse Q0
                           // Q1 = [ 0x0 0x8 0x4 0xC  0x2 0xA 0x6 0xE
                           // 0x1 0x9 0x5 0xD 0x3 0xB 0x7 0xF ]
VLDRB.S8      Q2, [R1, Q1] // Byte gather load, base in R1, offsets in Q1
```

3.21.20 VMOVL

Vector Move Long.

Syntax

```
VMOVL<T><v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the source element is used. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt This parameter determines the following values:
- `S8`
- `U8`
- `S16`
- `U16`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Selects an element of 8 or 16-bits from either the top half (T variant) or bottom half (B variant) of each source element, sign or zero-extends, performs a signed or unsigned left shift by an immediate value and places the 16 or 32-bit results in the destination vector.

VMOVL example

```
// 16-bit integer Vector Move Long (into 32-bit integer vector)
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1       // Generates incrementing sequence,
                               // starting at 0 with step of 1
VMUL.S16      Q0, Q0, r3       // Multiply by 1000
                               // Q0 = [0 1000 2000 3000 4000 5000 6000 7000]

// Move bottom parts (of adjacent pairs source)
VMOVLB.S16     Q1, Q0          // Q1[i] = Q0[2*i]        i={0..3}
                               // Q1 = [0 2000 4000 6000]

// Move top parts (of adjacent pairs source)
VMOVLT.S16     Q2, Q0          // Q2[i] = Q0[2*i+1]      i={0..3}
                               // Q2 = [1000 3000 5000 7000]
```

3.21.21 VMOVN

Vector Move and Narrow.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VMOVN<T><v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - `I16`
  - `I32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Performs an element-wise narrowing to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VMOVN example

```
// Merge 2 x 32-bit integer vectors into single 16-bit integer vector
MOV           R0, #0
MOV           R2, #1
MOV           R3, #1000
VIDUP.U32     Q0, R0, #2       // Generates incrementing sequence,
                               // starting at 0 with step of 2
VMUL.S32      Q1, Q0, R3       // Multiply by 1000
                               // Q1 = [ 0 2000 4000 6000 ]
VIDUP.U32     Q0, R2, #2       // Generates incrementing sequence,
                               // starting at 1 with step of 2
VMUL.S32      Q2, Q0, R3       // Multiply by 1000
                               // Q2 = [ 1000 3000 5000 7000 ]
// Move into bottom parts (of adjacent pairs destination)
VMOVNB.I32     Q0, Q1          // Q0[2*i] = Q1[i]   i={0..3}
                               // Q0 = [  0 x  2000 x  4000 x  6000 x  ]
                               // (x indicates the unchanged part of the vector)

// Move into top parts (of adjacent pairs destination)
VMOVNT.I32     Q0, Q2          // Q0[2*i+1] = Q2[i]  i={0..3}
                               // Q0 = [  x 1000  x 3000  x 5000  x 7000  ]
                               // (x indicates the unchanged part of the vector)
                               // Q0 = [  0 1000  2000 3000  4000 5000 6000 7000  ]
```

3.21.22 VQMOVN

Vector Saturating Move and Narrow.

Syntax

```
VQMOVN<T><v>.<dt> Qd, Qm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt This parameter determines the following values:
- `S16`
- `U16`
- `S32`
- `U32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VQMOVN example

```
// 32-bit integer Vector Saturating Move and Narrow.
// Merge 2 x 32-bit integer vectors into single 16-bit integer vector
VLDRW.32        Q1, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q2, [R1]    // 32-bit contiguous vector load
                            // Q1 = [ -4096 20480 -36864 53248 ]
                            // Q2 = [ 2560 12800 23040 33280 ]
// Move into bottom parts (of adjacent pairs destination)
VQMOVNB.s32     Q0, Q1      // Q0[2*i] = SSAT(Q1[i])      i={0..3}
                            // Q0 = [-4096 x 20480 x -32768 x 32767 x]
                            // (x indicates the unchanged part of the vector)
// Move into top parts (of adjacent pairs destination)
VQMOVNT.s32     Q0, Q2      // Q0[2*i+1] = SSAT(Q2[i])    i={0..3}
                            // Q0 = [ x 2560  x 12800  x 23040  x 32767 ]
                            // (x indicates the unchanged part of the vector)
                            // Q0 = [-4096 2560 20480 12800 -32768 23040 32767 32767]
```

3.21.23 VQMOVUN

Vector Saturating Move Unsigned and Narrow.

Syntax

```
VQMOVUN<T><v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value. The result is always saturated to an unsigned value.

VQMOVUN example

```
// 32-bit unsigned integer Vector Saturating Move and Narrow.
// Merge 2 x 32-bit unsigned integer vectors into single 16-bit unsigned integer vector
VLDRW.32        Q1, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q2, [R1]    // 32-bit contiguous vector load
                            // Q1 = [ -4096 20480 -36864 66666 ]
                            // Q2 = [ 2560 12800 23040 33280 ]
// Move into bottom parts (of adjacent pairs destination)
VQMOVUNB.S32   Q0, Q1       // Q0[2*i] = USAT(Q1[i])    i={0..3}
                            // Q0 = [ 0 x 20480 x 0 x 65535 x ]
                            // (x indicates the unchanged part of the vector)
// Move into top parts (of adjacent pairs destination)
VQMOVUNT.S32   q0, q2       // Q0[2*i+1] = USAT(Q2[i])       i={0..3}
                            // Q0 = [ x 2560 x 12800 x 23040 x 33280 ]
                            // (x indicates the unchanged part of the vector)
                            // Q0 = [ 0 2560 20480 12800 0 23040 65535
33280 ]
```

3.21.24 VQRSHL

Vector Saturating Rounding Shift Left.

Syntax

```
VQRSHL<v>.<dt> Qd, Qm, Qn
VQRSHL<v>.<dt> Qda, Rm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register, the elements of which containing the amount to shift by.

Rm Source general-purpose register containing the amount to shift by.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

`Rm` must not use the same register as `SP` and `PC`

Post-conditions

Updates QC flag in FPSCR register.

Operation

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

VQRSHL example

```
// 16-bit integer Vector Saturating Rounding Shift Left.
VLDRH.16      Q0, [R0]  // 16-bit vector contiguous load
                        // Q0 (in)  = [ 0x501 0xA01 0xF01 0x1401 0x1901 0x1E01 0x2301 
0x2801 ]
MOV           R0, #2    // left shift by 2
VQRSHL.S16    Q0,R0     // Q0[i] = SSAT(RSHIFT(Q0[i],2))
                        // Q0 (out) = [ 0x1404 0x2804 0x3C04 0x5004 0x6404 0x7804 0x7FFF 
0x7FFF ]

// scalar based right shift
VLDRH.16      Q0, [R0]  // 16-bit vector contiguous load
                        // Q0 (in)  = [ 0x501 0xA01 0xF01 0x1401 0x1901 0x1E01 0x2301 
0x2801 ]
mov           R1, #-2   // Right shift by 2
VQRSHL.S16    Q0,R1     // Q0[i] =  SSAT(RSHIFT(Q0[i],-2))
                        // Q0 (out) = [ 0x140 0x280 0x3C0 0x500 0x640 0x780 0x8C0 
0xA00 ]

// vector based shift
VLDRH.16      Q0, [R0]  // 16-bit vector contiguous load
                        // Q0 (in)  = [ 0x501 0xA01 0xF01 0x1401 0x1901 0x1E01 0x2301 
0x2801 ]
MOVS          R0, #1
MOVS          R1, #4
VIDUP.U16     Q1,R0,#1  // Generates incrementing sequence, starting at 1 with increments of 
1
VSUB.S16      Q1,Q1,R1  // Subtract 4 to Q1, generates Q1 = [ -3 -2 -1 0 1 2 3 4 ]
VQRSHL.S16    Q2,Q0,Q1  // Q2[i] = SSAT(RSHIFT(Q0[i],Q1[i]))   i={0..7}
                        // Q0 (in)  = [ 0x501 0xA01 0xF01 0x1401 0x1901 0x1E01 0x2301 
0x2801 ]
                        // Q1       = [ -3 -2 -1 0 1 2 3 4 ]  (shifts)
                        // Q0 (out) = [ 0xA0 0x280 0x781 0x1401 0x3202 0x7804 0x7FFF 
0x7FFF ]
```

3.21.25 VQRSHRN

Vector Saturating Rounding Shift Right and Narrow.

Syntax

```
VQRSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:

- S16
- U16
- S32
- U32

imm The number of bits to shift by, in the range 1 to dt/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field (sz).

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VQRSHRN example

```
// 32-bit integer Vector Saturating Rounding Shift Right and Narrow (into 16-bit integer
vector destination)
VLDRW.32      Q2, [R0]    // 32-bit contiguous vector load
VLDRW.32      Q3, [R1]    // 32-bit contiguous vector load
                          // Q2 = [ 0x10000001 0x20000001 0x30000001 0x40000001 ]
                          // Q3 = [ -0xFFFFFFF 0x30000001 -0x4FFFFFFF 0x70000001 ]
VMOV.S16      Q0, #0
VMOV.S16      Q1, #0
// Shift into bottom parts (of adjacent pairs destination)
VQRSHRNB.S32  Q0,Q2,#0xF  // Q0[2*i] = SSAT(RSHIFT_I(Q2[i], 15), 16)   i={0..3}
                          // Q0 = [ 0x2000 0x0 0x4000 0x0 0x6000 0x0 0x7FFF 0x0 ]
                          // Displaying the 128-bit register
                          // containing 16-bit-wide elements
// Shift into top parts (of adjacent pairs destination)
VQRSHRNT.S32  Q1,Q3,#0xF  // Q1[2*i+1] = SSAT(RSHIFT_I(Q3[i], 15), 16)   i={0..3}
                          // Q1 = [ 0x0 -0x2000 0x0 0x6000 0x0 -0x8000 0x0 0x7FFF ]
                          // Displaying the 128-bit register
                          // containing 16-bit-wide elements
// RSHIFT_I : Rounding Right Shift immediate
```

3.21.26 VQRSHRUN

Vector Saturating Rounding Shift Right Unsigned and Narrow.

Syntax

```
VQRSHRUN<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:

- B, indicates bottom half.
- T, indicates top half.

dt The size of the elements in the vector. This parameter must be one of the following values:

- S16.
- S32.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

imm The number of bits to shift by, in the range 1 to `dt`/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VQRSHRUN example

```
// Vector Saturating Rounding Shift Right Unsigned and Narrow
VLDRW.32      Q2, [R0]    // 32-bit contiguous vector load
VLDRW.32      Q3, [R1]    // 32-bit contiguous vector load
                          // Q2 = [ 0x10000001 0x20000001 0x30000001 0x40000001 ]
                          // Q3 = [ -FFFFFFF 0x30000001 -0x4FFFFFFF 0x70000001 ]
VMOV.S16        Q0, #0
VMOV.S16        Q1, #0
// shift into bottom parts (of adjacent pairs destination)
VQRSHRUNB.S32   Q0,Q2,#0xF  // Q0[2*i] = USAT(RSHIFT_I(Q2[i], 15), 16)   i={0..3}
                            // Q0 = [ 0x2000 0x0 0x4000 0x0 0x6000 0x0 0x8000 0x0 ]
// shift into top parts (of adjacent pairs destination)
VQRSHRUNT.S32   Q1,Q3,#0xF  // Q1[2*i+1] = USAT(RSHIFT_I(Q3[i], 15), 16)   i={0..3}
                            // Q1 = [ 0x0 0x0 0x0 0x6000 0x0 0x0 0x0 0xE000 ]
// RSHIFT_I : Rounding Right Shift immediate
```

3.21.27 VQSHL, VQSHLU

Vector Saturating Shift Left, Vector Saturating Shift Left Unsigned.

Syntax

```
VQSHL<v>.<dt> Qd, Qm, #<imm>
VQSHL<v>.<dt> Qd, Qm, Qn
VQSHL<v>.<dt> Qda, Rm
VQSHLU<v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register, the elements of which containing the amount to shift by.

Rm Source general-purpose register containing the amount to shift by.

dt This parameter determines the following values:

- `S8`
- `U8`
- `S16`
- `U16`
- `S32`
- `U32`

imm The number of bits to shift by, in the range 0 to `dt`-1. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

`Rm` must not use the same register as `SP` and `PC`

Post-conditions

Updates QC flag in FPSCR register.

Operation

The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

The immediate variant shifts each element of a vector register to the left by the immediate value.

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

The unsigned variant produces unsigned results, although the operands are signed.

VQSHL example

```
// Vector Saturating Shift Left
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
                             // Q0 = [ -0x300 0x900 -0xF00 0x1500 -0x1B00 0x2100 -0x2700
0x2D00 ]
// Immediate based left shift variant
VQSHL.S16       Q1, Q0, #2   // Q1[i] = SSAT(Q0[i] << 2)   i={0..7}
                             // Q1 = [ -0xC00 0x2400 -0x3C00 0x5400 -0x6C00
                             // 0x7FFF -0x8000 0x7FFF ]

// Register based left shift variant
MOV             R0, #2
                            // Q0 (in)  = [ -0x300 0x900 -0xF00 0x1500 -0x1B00 0x2100
-0x2700 0x2D00 ]
VQSHL.S16       Q0, R0      // Q0[i] = SSAT(Q0[i] << R0)   i={0..7}
                            // Q0 (out) = [ -0xC00 0x2400 -0x3C00 0x5400 -0x6C00
                            // 0x7FFF -0x8000 0x7FFF ]

// Vector based left shift variant
VLDRH.16        Q0, [R0]    // 16-bit vector contiguous load
                            // Q0 = [ -0x300 0x900 -0xF00 0x1500 -0x1B00 0x2100 -0x2700
0x2D00 ]
MOVS            R0, #1
MOVS            R1, #4
VIDUP.U16       Q1,R0,#1    // generates incrementing sequence, starting at 1 with
increments of 1
VSUB.S16        Q1,Q1,R1    // subtract 4 to incremented sequence [ -3 -2 -1 0 1 2 3 4 ]

VQSHL.S16       Q2, Q0, Q1  // Q2[i] = SSAT(Q0[i] << Q1[i])   i={0..7}
                            // Q1 = [ -3 -2 -1 0 1 2 3 4 ]
                            // Q2 = [ -0xC00 0x2400 -0x3C00 0x5400 -0x6C00
                            // 0x7FFF -0x8000 0x7FFF ]
```

VQSHLU example

```
// Vector Saturating Shift Left Unsigned
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
                             // Q0 = [  -0x300 0x900 -0xF00 0x1500 -0x1B00 0x2100 -0x2700
0x4E20 ]

VQSHLU.S16      Q1, Q0, #2   // Q1[i] = SSAT(Q0[i] << 2)     i={0..7}
                             // Q1 = [  0x0 0x2400 0x0 0x5400 0x0 0x8400 0x0 0x10000 ]
```

3.21.28 VQSHRN

Vector Saturating Shift Right and Narrow.

Syntax

```
VQSHRN<T><v>.<dt> Qd, Qm, #<imm>
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt This parameter determines the following values:
- `S16`
- `U16`
- `S32`
- `U32`

imm The number of bits to shift by, in the range 1 to `dt`/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VQSHRN example

```
// Vector Saturating Shift Right and Narrow
VLDRW.32       Q2, [R0]    // 32-bit contiguous vector load
VLDRW.32       Q3, [R1]    // 32-bit contiguous vector load
                           // Q2 = [ 0x10000001 0x20000001 0x30000001 0x40000001 ]
                           // Q3 = [ -0xFFFFFFF 0x30000001 -0x4FFFFFFF 0x70000001 ]
VMOV.S16       Q0, #0
VMOV.S16       Q1, #0

VQSHRNB.S32    Q0,Q2,#0xF  // Q0[2*i] = SSAT(SHIFT_I(Q2[i], 15) >> 16)
                           // Q0 = [ 0x2000 0x0 0x4000 0x0  0x6000 0x0  0x7FFF 0x0 ]

VQSHRNT.S32    Q1,Q3,#0xF  // Q1[2*i+1] = SSAT(SHIFT_I(Q3[i], 15) >> 16)
                           // Q1 = [ 0x0 -0x2000 0x0 0x6000 0x0 -0x8000 0x0 0x7FFF ]
```

3.21.29 VQSHRUN

Vector Saturating Shift Right Unsigned and Narrow.

Syntax

```
VQSHRUN<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt The size of the elements in the vector. This parameter must be one of the following values:
- `S16`.
- `S32`.

imm The number of bits to shift by, in the range 1 to `dt`/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Performs an element-wise saturation to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VQSHRUN example

```
// 32-bit Vector Saturating Shift Right Unsigned and Narrow
VLDRW.32        Q2, [r0]    // 32-bit contiguous vector load
VLDRW.32        Q3, [r1]    // 32-bit contiguous vector load
                            // Q2 = [ 0x10000001 0x20000001 0x30000001 0x40000001 ]
                            // Q3 = [ -0xFFFFFFF 0x30000001 -0x4FFFFFFF 0x70000001 ]
VMOV.S16        Q0, #0
VMOV.S16        Q1, #0

// shift into bottom parts (of adjacent pairs destination)
VQSHRUNB.S32    Q0, Q2, #15  // Q0[2*i] = USAT(SHIFT_I(Q2[i], 15) >> 16)     i={0..3}
                             // Q0 = [ 0x2000 0x0 0x4000 0x0 0x6000 0x0 0x8000 0x0 ]

// Shift into top parts (of adjacent pairs destination)
VQSHRUNT.S32    Q1, Q3, #15  // Q1[2*i+1] = USAT(SHIFT_I(Q3[i], 15) >> 16)   i={0..3}
                             // Q1 = [ 0x0 0x0 0x0 0x6000 0x0 0x0 0x0 E000 ]
```

3.21.30 VRSHL

Vector Rounding Shift Left.

Syntax

```
VRSHL<v>.<dt> Qd, Qm, Qn
VRSHL<v>.<dt> Qda, Rm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register, the elements of which containing the amount to shift by.

Rm Source general-purpose register containing the amount to shift by.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

The register variant shifts each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

VRSHL example

```
// Vector Rounding Shift Left.
VLDRH.16      Q0, [R0]    // 16-bit contiguous load, points to buffer containing
                          // {-0x2FD 0x903 -0xEFD 0x1503 -0x1AFD 0x2103 }
                          // Q0 (in)  = [ -0x2FD 0x903 -0x3837 -0xEFD -0x1AFD 0x2103 -0x26FD
0x2D03 ]
MOV           R0, #2

VRSHL.S16     Q0,R0       // Q0[i] = (Q0[i] << 2)    i = {0..7}
                          // Q0 (out) = [ -0x3060 0x240C -0x3BF4 0x540C -0x6BF4 -0x7BF4
0x640C -0x4BF4 ]
                          // Overflow occurs for the last three elements

// right shift (negative shift value)
MOV           R0, #-2     // Shift left by -2 (right shift)
                          // Q0 (in)  = [ -0x2FD 0x903 -0xEFD 0x1503 -0x1AFD 0x2103 -0x26FD
0x2D03 ]

VRSHL.S16     Q0,R0       // Q0[i] = ((Q0[i] + 2) >> 2)    i = {0..7}
                          // Q0 (out) = [ -0xBF 0x241 -0x3BF 0x541 -0x6BF 0x841 -0x9BF
0xB41 ]

// Vector based shift
MOVS          R0, #1
MOVS          R1, #4
VIDUP.U16     Q1,R0,#1    // Generates 16-bit incrementing sequence, starting at 1 with
increments of 1
VSUB.S16      Q1,Q1,R1    // Subtract 4 to incremented sequence [  -0x3 -0x2  -0x1 0x0  0x1
0x2  0x3 0x4]
                          // Q0  = [ -0x2FD 0x903  -0xEFD 0x1503  -0x1AFD 0x2103  -0x26FD
0x2D03]
                          // Q1  = [ -0x3 -0x2  -0x1 0x0  0x1 0x2  0x3 0x4  ]

VRSHL.S16     Q2,Q0,Q1    // Q2[i] = (Q0[i] + (1<<(Q1[i]-1)) << Q1[i] )  i = {0..7}
                          // Right shift for 1st half and left shift for 2nd half
                          // Q2  = [ -0x60 0x241  -0x77E 0x1503  -0x35FA -0x7BF4  -0x37E8
-0x2FD0  ] (Overflow occurs here)
```

3.21.31 VRSHR

Vector Rounding Shift Right.

Syntax

```
VRSHR<v>.<dt> Qd, Qm, #<imm>
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qd Destination vector register.

Qm Source vector register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

imm The number of bits to shift by, in the range 1 to dt. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

The variant shifts each element of a vector register to the right by the immediate value.

VRSHR example

```
// Vector Rounding Shift Right.
VLDRH.16      Q0, [R0]    // 16-bit load, points to buffer containing {-0x2FD 0x903 -0xEFD
0x1AFD -0x6909 0x2103 -0x26FD 0x2D03 }
                          // Q0 =[ -0x2FD 0x903 -0xEFD 0x1503 -0x1AFD 0x2103 -0x26FD
0x2D03 ]

VRSHR.S16     Q1, Q0, #5  // Q1[i] = (Q0[i] + (1 << 4)) >> 5    i = {0..7}
                          // Q1 =[ -0x18 0x48 -0x78 0xA8 -0xD8 0x108 -0x138 0x168 ]
```

3.21.32 VRSHRN

Vector Rounding Shift Right and Narrow.

Syntax

```
VRSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:

- B, indicates bottom half.
- T, indicates top half.

dt The size of the elements in the vector. This parameter must be one of the following values:

- I16.
- I32.

imm The number of bits to shift by, in the range 1 to dt/2. The encoding of this field is a logical OR of the most significant bits of the imm parameter and the least significant bits of the size field.

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Performs an element-wise narrowing to half-width, with shift, writing the rounded result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VRSHRN example

```
// Vector Rounding Shift Right and Narrow
VLDRH.16      Q0, [R0]      // 16-bit load, points to buffer containing
                            // {-0x2FD 0x903  -0xEFD 0x1503  -0x1AFD 0x2103  -0x26FD 0x2D03 }
                            // Q0 = [  -0x2FD 0x903  -0xEFD 0x1503  -0x1AFD 0x2103
                            // -0x26FD  0x2D03 ] (16-bit vector)
VLDRH.16      Q1, [R0, #16] // 16-bit load, points to buffer containing {-0x1FD 0x603
-0x9FD 0xE03
                            // -0x11FD 0x1603  -0x19FD 0x1E03 }
                            // Q1 = [  -0x1FD 0x603  -0x9FD 0xE03  -0x11FD 0x1603  -0x19FD
0x1E03  ]
                            // (16-bit vector)

// shift bottom parts (of adjacent pairs source)
VRSHRNB.I16   Q2,Q0,#7      // Q2[2*i] = (Q0[i] + (1 << 6)) >> 7    i = {0..7}
                            // Q2 = [  -0x6 x 0x12 x  -0x1E x 0x2A x  -0x36 x 0x42 x -0x4E x
0x5A x ]
                            // (8-bit vector, x indicates the unchanged part of the vector)


// shift top parts (of adjacent pairs source)
VRSHRNT.I16   Q2,Q1,#7      // Q2[2*i+1] = (Q1[i] + (1 << 6)) >> 7  i = {0..7} // merge into
Q2
                            // Q2 = [  -0x6 -0x4 0x12 0xC  -0x1E -0x14 0x2A 0x1C  -0x36
-0x24
                            // 0x42 0x2C  -0x4E -0x34 0x5A 0x3C ] (8-bit vector)
```

3.21.33 VSHL

Vector Shift Left.

Syntax

```
VSHL<v>.<dt> Qd, Qm, Qn
VSHL<v>.<dt> Qda, Rm
VSHL<v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register, the elements of which containing the amount to shift by.

Rm Source general-purpose register containing the amount to shift by.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:

- S8
- U8
- I8
- S16
- U16
- I16
- S32
- U32
- I32

imm The number of bits to shift by, in the range 0 to dt-1.

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

The immediate variant shifts each element of a vector register to the left by the immediate value.

The register variants shift each element of a vector register by the value specified in a source register. The direction of the shift depends on the sign of the element from the second vector register.

The vector variant shifts each element of the first vector by a value from the least significant byte of the corresponding element of the second vector and places the results in the destination vector.

VSHL example

```
// Vector Shift Left (or right with negative shift value)
VLDRH.16    Q0, [R0]    // 16-bit vector contiguous load
                        // Q0 = [ - 0x50 0xF0 -0x190 0x230 -0x2D0 0x370 -0x410 0x4B0 ]
// Immediate shift variant
VSHL.i16    Q1, Q0, #2  // Q1[i] = Q[0] << 2  i={0..7}
                        // Q1 = [ -0x140 0x3C0 -0x640 0x8C0 -0xB40 0xDC0 -0x1040 0x12C0 ]

// Register based variant
MOV         R0, #2
                        // Q0 = [ -0x50 0xF0 -0x190 0x230 -0x2D0 0x370 -0x410 0x4B0 ]
VSHL.S16    Q0, R0      // Q0[i] = Q[0] << R0  i={0..7}
                        // Q0 = [ -0x140 0x3C0 -0x640 0x8C0 -0xB40 DC0 -0x1040 0x12C0 ]

// Register based variant (negative left shift =right shift)
MOV         R0, #-2
                        // Q0 = [ -0x50 0xF0 -0x190 0x230 -0x2D0 0x370 -0x410 0x4B0 ]
VSHL.S16    Q0,R0       // Q0[i] = Q[0] >> R0  i={0..7}
                        // Q0 = [ -0x14 0x3C -0x64 0x8C -0xB4 0xDC -0x104 0x300 ]

// Vector based shift variant
MOVS          R0, #1
MOVS          R1, #4
VIDUP.U16     Q1,R0,#1  // Generates incrementing sequence, starting at 1 with increments of
1
VSUB.S16      Q1,Q1,R1  // Subtract 4 to incremented sequence [  -3 -2  -1 0  1 2  3 4  ]
                        // Q0 = [ -0x50 0xF0 -0x190 0x230 -0x2D0 0x370 -0x410 0x4B0 ]
                        // Q1 = [  -3 -2  -1 0  1 2  3 4  ]
VSHL.S16      Q2,Q0,Q1  // Q2[i] = Q[0] << Q1[i]     i={0..7}  (right shifts for 1st half,
left shift for 2nd half)
                        // Q2 = [ -A 0x3C -0xC8 0x230 -0x5A0 0xDC0 -0x2080 0x4B00 ]
```

3.21.34 VSHLC

Whole Vector Left Shift with Carry.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VSHLC<v> Qda, Rdm, #<imm>
```

Parameters

Qda Source and destination vector register.

Rdm Source and destination general-purpose register for carry in and out.

imm The number of bits to shift by, in the range 1-32.

v See Standard Assembler Syntax Fields

Restrictions

`Rdm` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Logical shift left by 1-32 bits, with carry across beats, carry in from general-purpose register, and carry out to the same general-purpose register. Permits treating a vector register as a single 128-bit scalar. The carry in is from the lower `imm` bits of the general-purpose register, not the upper bits.

VSHLC example

```
// Whole Vector Left Shift with Carry.
MOVW      R0, 0x4567
MOVT      R0, 0x0123             // R0 = 0x01234567
MOVW      R1, 0xCDEF
MOVT      R1, 0x89AB             // R1 = 0x89ABCDEF

// Vector init. using dual MOV
VMOV      Q0[2], Q0[0], R1, R1   // Q0 = [0x89ABCDEF x 0x89ABCDEF x]
VMOV      Q0[3], Q0[1], R0, R0   // Q0 = [0x89ABCDEF 0x01234567 0x89ABCDEF 0x01234567]
MOVW      R0, #0x2233
MOVT      R0, #0x0011            // R0 = 0x00112233

VSHLC     Q0, R0, #16            // Q0 = Q0 << 16 | (R0 >> 16)
                                 // Q0 =[ 0xCDEF2233 0x456789AB 0xCDEF0123 0x456789AB ]
                                 // = 0x456789ABCDEF0123456789ABCDEF2233

                                 // R0 = 0x00000123
```

3.21.35 VSHLL

Vector Shift Left Long.

Syntax

```
VSHLL<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the source element is used. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:
- S8
- U8
- S16
- U16

imm The number of bits to shift by, in the range 1 to `dt`.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Selects an element of 8 or 16-bits from either the top half (T variant) or bottom half (B variant) of each source element, performs a left shift by an immediate value, performs a signed or unsigned left shift by an immediate value and places the 16 or 32-bit results in the destination vector.

VSHLL example

```
// 16-bit Vector Shift Left Long (into 32-bit vector desrtination)
VLDRH.16    Q0, [R0]    // 16-bit vector contiguous load
                        // Q0 = [  0x1001 0x2001  0x3001 0x4001 0x5001 0x6001 0x7001
-0x7FFF  ]

// Shift bottom parts (of adjacent pairs source)
VSHLLB.S16  Q1,Q0,#2    // Q1[i] = Q0[2*i] << 2     i={0..3}
                        // Q1 = [  0x4004 0xC004 0x14004 0x1C004 ]

// Shift top parts (of adjacent pairs source)
                        // Q0 = [  0x1C004 0x2001  0x3001 0x4001  0x5001 0x6001  0x7001
-0x7FFF  ]
VSHLLT.S16  Q2,Q0,#2    // Q2[i] = Q0[2*i+1] << 2   i={0..3}
                        // Q2 = [  0x8004 0x10004 0x18004 -0x1FFFC ]
```

3.21.36 VSHR

Vector Shift Right.

Syntax

```
VSHR<v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt This parameter determines the following values:
- S8
- U8
- S16
- U16
- S32
- U32

imm The number of bits to shift by, in the range 1 to `dt`.

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Shifts each element of a vector register to the right by the immediate value.

VSHR example

```
// 16-bit Vector Shift Right.
VLDRH.16    Q0, [R0]    // 16-bit vector contiguous load
                        // Q0 = [ -0x50 0xF0  -0x190 0x230  0x-2D0 0x370  -0x410 0x4B0  ]

VSHR.S16    Q1, Q0, #2  // Q1[i] = Q0[i] >>  2     i={0..7}
                        // Q1 = [ -0x14 0x3C  -0x64 0x8C  -0xB4 0xDC  -0x104 0x12C  ]
```

3.21.37 VSHRN

Vector Shift Right and Narrow.

Syntax

```
VSHRN<T><v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies which half of the result element the result is written to. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt The size of the elements in the vector. This parameter must be one of the following values:
- `I16`.
- `I32`.

imm The number of bits to shift by, in the range 1 to `dt`/2.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Performs an element-wise narrowing to half-width, with shift, writing the result to either the top half (T variant) or bottom half (B variant) of the result element. The other half of the destination vector element retains its previous value.

VSHRN example

```
// Vector Shift Right and Narrow
MOV             R0, #1
MOV             R1, #1000
VIDUP.U32       Q0, R0, #1    // Generates 32-bit incrementing sequence, starting at 1 with
increments of 1
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMUL.S32        Q0, Q0, r1    // Multiply by 1000
VMOV.S16        Q2, #0
// Shift into bottom parts (of adjacent pairs destination)
                              // Q0 = [ 0x3E8 0x7D0 0xBB8 0xFA0 ]  (32-bit vector)
VSHRNB.I32      Q2, Q0, #2    // Q2[2*i] = Q0[i] >> 2   i = {0..3}
                              // Q2 = [  0xFA 0x0  0x1F4 0x0  0x2EE 0x0  0x3E8 0x0  ] (16-
bit vector)

// Shift into top parts (of adjacent pairs destination)
VLDRW.32        Q1, [R2]      // 32-bit contiguous vector load
                              // Q1 = [ -0x3E8 0xBB8 -0x1388 0x1B58 ] (32-bit vector)
VMOV.S16        Q2, #0
VSHRNT.I32      Q2, Q1, #2    // Q2[2*i+1] = Q0[i] >> 2  i = {0..3}
                              // Q2 = [  0x0 -0xFA  0x0 0x2EE  0x0 -0x4E2  0x0 0x6D6  ] (16-
bit vector)

// Merging
VMOV.S16        Q2, #0
                              // Q0 = [ 0x3E8 0x7D0 0xBB8 0xFA0 ]  (32-bit vector)
                              // Q1 = [ -0x3E8 0xBB8 -0x1388 0x1B58 ] (32-bit vector)
VSHRNB.I32      Q2, Q0, #2    // Q2[2*i]   = Q0[i] >> 2  i = {0..3}
VSHRNT.I32      Q2, Q1, #2    // Q2[2*i+1] = Q1[i] >> 2  i = {0..3}
                              // Q2 = [ 0xFA -0xFA  0x1F4 0x2EE  0x2EE -0x4E2  0x3E8
0x6D6  ] (16-bit vector)
```

3.21.38 VSLI

Vector Shift Left and Insert.

Syntax

```
VSLI<v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt The size of the elements in the vector. This parameter must be one of the following values:

- 8.
- 16.
- 32.

imm The number of bits to shift by, in the range 0 to `dt`-1.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Takes each element in the operand vector, left shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the left of each element are lost.

VSLI example

```
// Vector Shift Left and Insert.
MOV           R0, #1        // Incrementing sequence start
MOV           R1, #0x2112   // Random sequence multiplier
MOV           R2, #1        // Incrementing sequence start
MOV           R3, #0x656    // Random sequence multiplier
VIDUP.u16     Q0, R0, #2    // Generates incrementing sequence, starting at 1 with
increments of 2
VMUL.s16      Q0, Q0, R1    // Q0[i] = Q0[i] * 0x2112 i={0..7}
VIDUP.u16     Q1, R2, #4    // Generates incrementing sequence, starting at 1 with
increments of 4
                            // Q0 (in) = [ 0x2112 0x6336 0xA55A 0xE77E 0x29A2 0x6BC6c 0xADEA
0xF00E ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMUL.s16      Q1, Q1, R3    // Q1[i] = Q1[i] * 0x656  i={0..7}
                            // Q0 (in) = [ 0x2112 0x6336 0xA55A 0xE77E 0x29A2 0x6BC6 0xADEA
0xF00E ]
                            // Q1      = [ 0x0656 0x1FAE 0x3906 0x525E 0x6BB6 0x850E 0x9E66
0xB7BE ]

// Shift Left Q1 by 4 and Insert Q0 4-bits LSBs
VSLI.16       Q0, Q1, #4    // Q0[i] = Q1[i] << 4 | Q0[i] & 0xF i={0..7}
                            // Q0 (out)= [ 0x6562 0xFAE6 0x906A 0x25EE 0xBB62 0x50E6 0xE66A
0x7BEE ]

// 32-bit variant
MOV           R0, #1        // Incrementing sequence start
MOVW          R1, #0x1112   // Random sequence multiplier
MOVT          R1, #0x2111
MOV           R2, #4        // Incrementing sequence start
MOVW          R3, #0x1113   // Random sequence multiplier
MOVT          R3, #0x3111
VIDUP.u32     Q0, R0, #1    // Generates incrementing sequence, starting at 1 with
increments of 1
VMUL.s32      Q0, Q0, R1    // Q0[i] = Q0[i] * 0x21121112    i={0..3}
VIDUP.u32     Q1, R2, #1    // Generates incrementing sequence, starting at 4 with
increments of 1
VMUL.s32      Q1, Q1, R3    // Q1[i] = Q1[i] * 0x31111113    i={0..3}
                            // Q0 (in) = [ 0x21111112 0x42222224 0x63333336 0x84444448 ]
                            // Q1      = [ 0xC444444C 0xF555555F 0x26666672 0x57777785 ]

// Shift Left Q1 by 12 and Insert Q0 12-bits LSBs
VSLI.32       Q0, Q1, #12   // Q0[i] = Q1[i] << 12 | Q0[i] & 0xFFF  i={0..3}
                            // Q0 (out)= [ 0x4444C112 0x5555F224 0x66672336 0x77785448 ]
```

3.21.39 VSRI

Vector Shift Right and Insert.

Syntax

```
VSRI<v>.<dt> Qd, Qm, #<imm>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt The size of the elements in the vector. This parameter must be one of the following values:
- 8.
- 16.
- 32.

imm The number of bits to shift by, in the range 1 to `dt`.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Takes each element in the operand vector, right shifts them by an immediate value, and inserts the results in the destination vector. Bits shifted out of the right of each element are lost.

VSRI example

```
// Vector Shift Right and Insert
MOV           R0, #1         // Incrementing sequence start
MOV           R1, #0x2112    // Random sequence multiplier
MOV           R2, #1        // Incrementing sequence start
MOV           R3, #0x656     // Random sequence multiplier
VIDUP.U16     Q0, R0, #2    // Generates incrementing sequence, starting at 1 with
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
increments of 2
VMUL.S16      Q0, Q0, r1    // Q0[i] = Q0[i] * 0x2112 i={0..7}
VIDUP.U16     Q1, R2, #4    // Generates incrementing sequence, starting at 1 with
increments of 4
VMUL.S16      Q1, Q1, r3    // Q0[i] = Q0[i] * 0x656  i={0..7}
                            // Q0 (in) = [ 0x2112 0x6336 0xA55A 0xE77E 0x29A2 0x6BC6 0xADEA
0xF00E ]
                            // Q1      = [ 0x0656 0x1FAE 0x3906 0x525E 0x6BB6 0x850E 0x9E66
0xB7BE ]

VSRI.16       Q0, Q1, #4    // Q0[i] = Q1[i] >> 4 | Q0[i] & 0xF000 i={0..7}
                            // Q0 (out)= [ 0x2065 0x61FA 0xA390 0xE525 0x26BB 0x6850 0xA9E6
0xFB7B  ]

MOV           R0, #1        // Incrementing sequence start
MOVW          R1, #0x1112   // Random sequence multiplier
MOVT          R1, #0x2111
MOV           R2, #4        // Incrementing sequence start
MOVW          R3, #0x1113   // Random sequence multiplier
MOVT          r3, #0x3111
VIDUP.U32     Q0, R0, #1    // Generates incrementing sequence, starting at 1 with
increments of 1
VMUL.S32      Q0, Q0, R1    // Q0[i] = Q0[i] * 0x21121112 i={0..3}
VIDUP.U32     Q1, R2, #1    // Generates incrementing sequence, starting at 4 with
increments of 1
VMUL.S32      Q1, Q1, R3    // Q0[i] = Q0[i] * 0x31111113 i={0..3}
                            // Q0 (in) = [ 0x21111112 0x42222224 0x63333336 0x84444448  ]
                            // Q1      = [ 0xC444444C 0xF555555F 0x26666672 0x57777785 ]

VSRI.32       Q0, Q1, #12   // Q0[i] = Q1[i] >> 12 | Q0[i] & 0xFFF00000 i={0..3}
                            // Q0 (out)= [ 0x211C4444 0x422F5555 0x63326666 0x84457777 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.22 Arm®v8.1-M branch and loop instructions

Reference material for the Cortex-M55 processor Armv8.1-M branch and loop operations instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.22.1 List of Arm®v8.1-M branch and loop instructions

An alphabetically ordered list of the Armv8.1-M comparison and vector predication operations instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-16 Armv8.1-M branch and loop instructions

Mnemonic	Brief description	See
`BF, BFX, BFL, BFLX, BFCSEL`	Branch Future, Branch Future and Exchange, Branch Future with Link, Branch Future with Link and Exchange, Branch Future Conditional Select.	*3.22.2 BF, BFX, BFL, BFLX, BFCSEL* on page 3-295
`LCTP`	Loop Clear with Tail Predication	*3.22.3 LCTP* on page 3-297
`LE, LETP`	Loop End, Loop End with Tail Predication	*3.22.4 LE, LETP* on page 3-297
`WLS, DLS, WLSTP, DLSTP`	While Loop Start, Do Loop Start, While Loop Start with Tail Predication, Do Loop Start with Tail Predication	*3.22.5 WLS, DLS, WLSTP, DLSTP* on page 3-298

3.22.2 BF, BFX, BFL, BFLX, BFCSEL

Branch Future, Branch Future and Exchange, Branch Future with Link, Branch Future with Link and Exchange, Branch Future Conditional Select.

Syntax

```
BFX<c> <b_label>, Rn
BFL<c> <b_label>, <label>
BFCSEL <b_label>, <label>, <ba_label>, <bcond>
BF<c> <b_label>, <label>
BFLX<c> <b_label>, Rn
```

Parameters

`Rn` The address to branch to.

`T` Selects whether the instruction at `b_label` is a 2-byte (T = 0) or 4-byte (T = 1) instruction to be branched around, as specified by `ba_label`.

`b_label` The PC relative offset of the first instruction in the fallback code, that will not be executed if the future branch is taken.

`ba_label` The PC relative offset of the address to branch to in case the associated BFCSEL condition code check fails and no other branch future is pending. The range of this address allows branching over a 2-byte or 4-byte instruction located at `b_label`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

bcond The comparison condition to use. The evaluation of this comparison is performed when this instruction is executed and not at the point the branch is performed. This parameter must be one of the following values:

- EQ.
- NE.
- CS.
- CC.
- MI.
- PL.
- VS.
- VC.
- HI.
- LS.
- GE.
- LT.
- GT.
- LE.

c See Standard Assembler Syntax Fields

label The PC relative offset of the address to branch to.

Restrictions

Rn must not use the same register as SP and PC.

Post-conditions

There are no condition flags.

Operation

These instructions behave as NOP in Cortex-M55.

BF example

```
// BF sample
  BF     #1F, __myfunc
// random code...
1:                         // effective branch point
  b      __myfunc
```

BFX example

```
// Branch Future and Exchange (BFX) sample
  BFX     #1F, LR
// random code...
1:
  BX      LR
```

BFL example

```
// Branch Future with Link (BFL) sample
  BFL     #1F, __myfunc
// random code...
1:
  bl      __myfunc
```

BFCSEL example

```
// Branch Future Conditional Select (BFCSEL) sample using condition
3:
// random code...
    CMP     R0, R1
// random code
    BFCSEL  #1F, #3B, #2F, NE
// random code...
1:
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
    BNE.W   #3B
2:
```

3.22.3 LCTP

Loop Clear with Tail Predication.

For more information on low-overhead loop operation, see 3.22.5 WLS, DLS, WLSTP, DLSTP on page 3-298.

Syntax

```
LCTP<c>
```

Parameters

c See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Exits loop mode by invalidating LO_BRANCH_INFO and clears any tail predication being applied.

LCTP example

```
// Low overhead loop with tail predication break
MOV             R0, #0
MOV             R2, #0
MOV             R1, #0xA
VIDUP.U32       Q0, R0, #1   // 32-bit incrementing sequence starting at 0, increment of 1
DLSTP.32        LR, R1       // Loop + Tail predication start
1:
VADDVA.S32      R2, Q0       // R2 = R2 + sum(Q0[i])   i={0..3}
CMP             LR, #5       // Compare loop elements
ITT             LE
MOVLE           LR, #0       // Clear loop elements counter
BLE             #2F          // Jump out of the loop
VIDUP.U32       Q0, R0, #1   // Incrementing sequence continuation, increment of 1
LETP            LR, #1B      // Loop + Tail predication end

2:
LCTP                        // Clear Loop context and Tail Predication
VADD.S32        Q0, Q0, Q1  // Safe MVE execution after loop break
```

3.22.4 LE, LETP

Loop End, Loop End with Tail Predication.

Syntax

```
LE <label>
LE LR, <label>
LETP LR, <label>
```

Parameters

LR LR is used to hold the iteration counter of the loop, and these instructions must always use this register.

label Specifies the label of the first instruction in the loop body.

Restrictions

There are no restrictions.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

If additional iterations of a loop are required, this instruction branches back to the `label`. It also stores the loop information in the loop info cache so that future iterations of the loop will branch back to the start just before the `LE` instruction is encountered. The first variant of the instruction checks a loop iteration counter (stored in LR) to determine if additional iterations are required. It also decrements the counter ready for the next iteration.

The second variant does not use an iteration count and always triggers another iteration of the loop.

The third variant also checks the loop iteration counter to determine if additional iterations are required. However the counter is decremented by the number of elements in a vector (as indicated by the FPSCR.LTPSIZE field). On the last iteration of the loop, this variant disables tail predication. For more information on FPSCR, see *4.12.5 Floating-point Status Control Register, FPSCR* on page 4-618.

LE example

```
// LE
MOV                 LR, #3
WLS                 LR, LR, 1F          // Loop start (LR contains loop occurrences)
2:
VLDRH.16            Q0, [R2], #16       // Load source matrix row
VLDRH.S16           Q1, [R3], #16       // Load source vector row
VMLALDAVA.S16       R0, R1, Q0, Q1      // 16-bit accumulated dot-product
LE                  LR, 2B              // Loop end
1:
```

LETP example

```
// LETP
MOV                 LR, #15
WLSTP.16            LR, LR, 1F          // Loop + tail predication start (lr contains
samples to process)
2:
vldrh.16            Q0, [R2], #16       // Load source matrix row
vldrh.s16           Q1, [R3], #16       // Load source vector row
vmlaldava.s16       R0, R1, Q0, Q1      // 16-bit accumulated dot-product
letp                lR, 2B              // Loop + tail predication end
1:
```

3.22.5 WLS, DLS, WLSTP, DLSTP

While Loop Start, Do Loop Start, While Loop Start with Tail Predication, Do Loop Start with Tail Predication.

Syntax

```
DLSTP.<size> LR, Rn
DLS LR, Rn
WLS LR, Rn, <label>
WLSTP.<size> LR, Rn, <label>
```

Parameters

`LR` For the `WLSTP` and `DLSTP` variants, LR is used to hold the number of elements to process. These instructions must always use this register. For the `WLS` and `DLS` variants, LR is used to hold the iteration counter of the loop, this instruction must always use this register.

`Rn` For the `WLSTP` and `DLSTP` variants, this is the register holding the number of elements to process. For the `WLS` and `DLS` variants, this is the register holding the number of loop iterations to perform.

`label` For the `WLSTP` variant, this specifies the label of the instruction after the loop (the first instruction after the `LE`). For the `WLS` variant, this specifies the label of the instruction to branch to if no loop iterations are required.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

size The size of the elements in the vector to process. This value is stored in the FPSCR.LTPSIZE field, and causes tail predication to be applied on the last iteration of the loop. This parameter must be one of the following values:

- 8
- 16
- 32
- 64

Restrictions

- `Rn` must not use the same register as `SP` and `PC`.
- `Rn` must not use `SP`.

Post-conditions

There are no condition flags.

Operation

This instruction partially sets up a loop. An `LE` or `LETP` (Loop End) instruction completes the setup. The base variants of this instruction (`WLS` and `DLS`) set `LR` to the number of loop iterations to be performed, whereas the other variants of this instruction set `LR` to the number of vector-elements that must be processed. For the other variants, if the number of elements required is not a multiple of the vector length then the appropriate number of vector elements will be predicated on the last iteration of the loop. When using `WLS` or `WLSTP`, if the number of iterations required is zero, then these instructions branch to the label specified. Each loop start instruction is normally used with a matching `LE` or `LETP` instruction.

WLS example

```
//WLS
MOV             LR, #0xF         // Number of loop iterations
WLS             LR, LR, 1F      // While loop start
2:
  LDRH          R2, [R0], #2    // Scalar half word load with post increment
  STRH          R2, [R1], #2    // Scalar half word store with post increment
LE              LR, 2B          // Loop end
1:
```

WLSTP example

```
// WLSTP
MOV             LR, #15         // Number of vector elements to process
WLSTP.8         LR, LR, 1F      // While loop start with tail-predication
2:
  VLDRB.S8      Q0, [R0], #16   // 8-bit contiguous vector load
  VSTRB.S8      Q0, [R1], #16   // 8-bit contiguous vector store
LETP            LR, 2B          // Loop with tail-predication end
1:
```

DLS example

```
// DLS
MOV             R2, #10          // Loop occurrences
DLS             LR, R2           // Do loop start trigger (iteration count in R2)
1:
VLDRW.32        Q0, [R0, #16]!   // Vector contiguous load + post-increment
VRINTP.F32      Q0, Q0           // Round
VSTRW.32        Q0, [R1, #16]!   // Vector contiguous store + post-increment
LE              LR,  #1B         // Loop end
```

DLSTP example

```
// DLSTP
// Count elements from 0 to 10 excluded
MOV             R0, #0    // Incrementing sequence start
MOV             R2, #0    // Accumulator initialization
MOV             R1, #10   // Vector elements to process (we want sum [0..9])
VIDUP.U32       Q0,R0,#1  // Incrementing sequence starting at 0, increment of 1
DLSTP.32        LR, R1    // Do loop + tail predication start, number of 32-bit elements in
r1
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
1:
VADDVA.S32      R2, Q0    // R2 = R2 + sum(Q0[i])      i={0..3}
VIDUP.U32       Q0,R0,#1  // Incrementing sequence continuation, increment of 1
LETP            LR, 1B    // Loop + tail-predication end
                          // R2 = 45 = sum [0..9]
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.23 Arm®v8.1-M comparison and vector predication operations instructions

Reference material for the Cortex-M55 processor Armv8.1-M comparison and vector predication operations instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.23.1 List of Arm®v8.1-M comparison and vector predication operations instructions

An alphabetically ordered list of the Armv8.1-M comparison and vector predication operations instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-17 Armv8.1-M comparison and vector predication operations instructions

Mnemonic	Brief description	See
`CINC`	Conditional Increment	*3.24.2 VLD2* on page 3-325
`CINV`	Conditional Invert	*3.24.3 VLD4* on page 3-326
`CNEG`	Conditional Negate	*3.20.13 VLDR* on page 3-233
`CSEL`	Conditional Select	*3.24.5 VLDRB, VLDRH, VLDRW* on page 3-329
`CSET`	Conditional Set	*3.24.6 VLDRB, VLDRH, VLDRW, VLDRD (vector)* on page 3-330
`CSETM`	Conditional Set Mask	*3.24.7 VST2* on page 3-332
`CSINC`	Conditional Select Increment	*3.24.8 VST4* on page 3-333
`CSINV`	Conditional Select Invert	*3.20.33 VSTR* on page 3-253
`CSNEG`	Conditional Select Negation	*3.24.10 VSTRB, VSTRH, VSTRW* on page 3-335
`VCMP`	Vector Compare	*3.23.11 VCMP* on page 3-309
`VCMP`	`VCMP` (floating-point) Vector Compare	*3.23.12 VCMP (floating-point)* on page 3-310
`VCTP`	Create Vector Tail Predicate	*3.23.13 VCTP* on page 3-311
`VMAX, VMAXA`	Vector Maximum, Vector Maximum Absolute	*3.23.14 VMAX, VMAXA* on page 3-312
`VMAXNM, VMAXNMA`	`VMAXNM, VMAXNMA` (floating-point) Vector Maximum, Vector Maximum Absolute	*3.23.15 VMAXNM, VMAXNMA (floating-point)* on page 3-313
`VMAXNMV, VMAXNMAV`	`VMAXNMV, VMAXNMAV` (floating-point) Vector Maximum Across Vector, Vector Maximum Absolute Across Vector	*3.23.16 VMAXNMV, VMAXNMAV (floating-point)* on page 3-314
`VMAXV, VMAXAV`	Vector Maximum Across Vector, Vector Maximum Absolute Across Vector	*3.23.17 VMAXV, VMAXAV* on page 3-314
`VMIN, VMINA`	Vector Minimum, Vector Minimum Absolute	*3.23.18 VMIN, VMINA* on page 3-315

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-17 Armv8.1-M comparison and vector predication operations instructions (continued)

Mnemonic	Brief description	See
VMINNM, VMINNMA	VMINNM, VMINNMA (floating-point) Vector Minimum, Vector Minimum Absolute	*3.23.19 VMINNM, VMINNMA (floating-point)* on page 3-316
VMINNMV, VMINNMAV	VMINNMV, VMINNMAV (floating-point) Vector Minimum Across Vector, Vector Minimum Absolute Across Vector	*3.23.20 VMINNMV, VMINNMAV (floating-point)* on page 3-318
VMINV, VMINAV	Vector Minimum Across Vector, Vector Minimum Absolute Across Vector	*3.23.21 VMINV, VMINAV* on page 3-318
VPNOT	Vector Predicate NOT	*3.23.22 VPNOT* on page 3-319
VPST	Vector Predicate Set Then	*3.23.23 VPST* on page 3-320
VPT	Vector Predicate Then	*3.23.24 VPT* on page 3-321
VPT	VPT (floating-point) Vector Predicate Then	*3.23.25 VPT (floating-point)* on page 3-323

3.23.2 CINC

Conditional Increment.

Syntax

```
CINC Rd, Rn, <fcond>
is equivalent to
CSINC Rd, Rn, Rn, invert (<cond>)
and is the preferred disassembly when <Rn == Rm && Rn != 15>
```

Parameters

None.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the value of the source register incremented by 1, if the condition is TRUE. Otherwise, returns the value of the source register.

This is an alias of CSINC with the following condition satisfied: `Rn == Rm && Rn != 15`.

CINC example

```
// Conditional increment of R0 based on 2 different conditions
MOV         R0, #0x1000
MOV         R1, #0x2000
CMP         R0, R1
CINC        R2, R0, LE    // If(cond == LE) R2 = R0 + 1
                          // The condition is TRUE in this case, therefore, R2 = 0x1001

CINC        R3, R0, GE    // If(cond == GE) R3 = R0 + 1
                          // The condition is FALSE in this case, therefore, R3 = 0x1000
```

3.23.3 CINV

Conditional Invert.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
CINV Rd, Rn, <fcond>
is equivalent to
CSINV Rd, Rn, Rn, invert (<cond>)
and is the preferred disassembly when <Rn == Rm && Rn != 15
```

Parameters

None.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the bitwise inversion of the value of the source register, if the condition is TRUE. Otherwise returns the value of the source register.

This is an alias of CSINV with the following condition satisfied: `Rn == Rm && Rn != 15`.

CINV example

```
// Conditional invert of R0 based on 2 different conditions
MOV         R0, #0x1000
MOV         R1, #0x2000
CMP         R0, R1
CINV        R2, R0, LE    // if(cond == LE) R2 = ~R0
                          // The condition is TRUE in this case, therefore, R2 = 0xFFFFEFFF
(= -0x1001)
CINV        R3, R0, GE    // if(cond == GE) R3 = ~R0
                          // The condition is FALSE in this case, therefore, R3 = 0x00001000
```

3.23.4 CNEG

Conditional Negate.

Syntax

```
CNEG Rd, Rn, <fcond>
is equivalent to
CSNEG Rd, Rn, Rn, invert (<cond>)
and is the preferred disassembly when <Rn == Rm>
```

Parameters

None.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the negated value of the source register if the condition is TRUE, and otherwise returns the value of the source register.

This is an alias of CSNEG with the following condition satisfied: `Rn == Rm`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CNEG example

```
// Conditional negate of R0 based on 2 different conditions
MOV         R0, #4096
MOV         R1, #8192
CMP         R0, R1
CNEG        R2, R0, LE    // if(cond == LE) R2 = -R0
                          // The condition is TRUE in this case, therefore, R2 = 4096

CNEG        R3, R0, GE    // if(cond == GE) R3 = -R0
                          // The condition is FALSE in this case, therefore, R3 = 4096
```

3.23.5 CSEL

Conditional Select.

Syntax

```
CSEL Rd, Rn, Rm, <fcond>
```

Parameters

Rd Destination general-purpose register.

Rm Second source general-purpose register (ZR is permitted, PC is not). ZR is the zero register and behaves as RAZ/WI.

Rn First source general-purpose register (ZR is permitted, PC is not).

fcond The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values:

- EQ.
- NE.
- CS.
- CC.
- MI.
- PL.
- VS.
- VC.
- HI.
- LS.
- GE.
- LT.
- GT.
- LE.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register.

CSEL example

```
// Conditional negate of r0 based on 2 different conditions
MOV         R0, #4096
MOV         R1, #8192
CMP         R0, R1
CNEG        R2, R0, LE    // if(cond == LE) R2 = -R0
                          // In thi case, the condition is TRUE, therefore, R2 = -4096
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CNEG        R3, R0, GE    // if(cond == GE) R3 = -R0
                          // In this case, the condition is FALSE, therefore, R3 = 4096
```

3.23.6 CSET

Conditional Set.

Syntax

```
CSET Rd, <fcond>
is equivalent to
CSINC Rd, Zr, Zr, invert (<cond>)
and is the preferred disassembly when <Rn == 0xF && Rm == 0xF>
```

Parameters

None.

Restrictions

`Rd` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

Operation

Sets the destination register to 1 if the condition is TRUE, and otherwise set it to 0.

This is an alias of CSINC with the following condition satisfied: `Rn==0xF && Rm==0xF`.

CSET example

```
// Conditional set of R2 and R3 based on two different conditions
MOV         R0, #4096
MOV         R1, #8192
CMP         R0, R1

CSET        R2, LE              // R2 = (cond == LE) ? 1 : 0
                                // In this case, the condition is TRUE, therefore R2 = 1

CSET        R3, GE              // R3 = (cond == GE) ? 1 : 0
                                // In this case, the condition is FALSE, therefore, R3 = 0
```

3.23.7 CSETM

Conditional Set Mask.

Syntax

```
CSETM Rd, <fcond>
is equivalent to
CSINV Rd, Zr, Zr, invert (<cond>)
and is the preferred disassembly when <Rn == 0xF && Rm == 0xF>
```

Parameters

None.

Restrictions

`Rd` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Sets all bits of the destination register to 1 if the condition is TRUE. Otherwise sets all bits to 0.

This is an alias of CSINV with the following condition satisfied: `Rn==0xF && Rm==0xF`.

CSETM example

```
//Conditional set mask of R2 and R3 based on two different conditions
MOV         R0, #4096
MOV         R1, #8192
CMP         R0, R1

CSETM       R2, LE               // R2 = (cond == LE) ? 0xFFFFFFFF : 0
                                 // In this case, the condition is TRUE, so R2 = 0xFFFFFFFF

CSETM       R3, GE               // r3 = (cond == GE) ? 0xFFFFFFFF : 0
                                 // In this case, the condition is FALSE, so R3 = 0
```

3.23.8 CSINC

Conditional Select Increment.

Syntax

```
CSINC Rd, Rn, Rm, <fcond>
```

Parameters

Rd Destination general-purpose register.

Rm Second source general-purpose register (ZR is permitted, PC is not).

Rn First source general-purpose register (ZR is permitted, PC is not).

fcond The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values:

- EQ.
- NE.
- CS.
- CC.
- MI.
- PL.
- VS.
- VC.
- HI.
- LS.
- GE.
- LT.
- GT.
- LE.

Restrictions

- `Rd` must not use the same register as `SP` and `PC`
- `Rn` must not use `SP`

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register incremented by 1.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

CSINC example

```
// Conditional Select Increment.
EOR         R0, R0, R0
MOV         R1, #10
                            // R0 == 0 ? R3 = R1 + 1 : R4 = R1 - 1
CMP         R0, #0
SUB         R2, R1, #1      // R2 = R1 - 1

CSINC       R3, R2, R1, NE  // R3 = (cond == NE) ? R2 : R1 + 1
                            // In this case, the condition is TRUE, therefore, R3 = 11

CSINC       R4, R2, R1, GE  // R4 = (cond == GE) ? R2 : R1 + 1
                            // In this case, the condition is FALSE, therefore, R4 = 9
```

3.23.9 CSINV

Conditional Select Invert.

Syntax

```
CSINV Rd, Rn, Rm, <fcond>
```

Parameters

Rd Destination general-purpose register.

Rm Second source general-purpose register (ZR is permitted, PC is not). ZR is the zero register and behaves as RAZ/WI.

Rn First source general-purpose register (ZR is permitted, PC is not).

fcond The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values:

- EQ.
- NE.
- CS.
- CC.
- MI.
- PL.
- VS.
- VC.
- HI.
- LS.
- GE.
- LT.
- GT.
- LE.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register, bitwise inverted.

CSINV example

```
// Conditional selection & inversion of R2 and R3 based on two different conditions
MOV         R0, #0x1000
MOV         R1, #0x2000
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CMP         R0, R1

CSINV       R2, R1, R0, LE    // R2 = (cond == LE) ? R1 : ~R0
                              // In this case, the condition is TRUE,
                              // therefore, R2 = 0x00002000

CSINV       R3, R1, R0, GE    // R3 = (cond == GE) ? R1 : ~R0
                              // In this case, the condition is FALSE,
                              // therefore, R3 = 0xFFFFEFFF
```

3.23.10 CSNEG

Conditional Select Negation.

Syntax

```
CSNEG Rd, Rn, Rm, <fcond>
```

Parameters

Rd Destination general-purpose register.

Rm Second source general-purpose register (ZR is permitted, PC is not). ZR is the zero register and behaves as RAZ/WI.

Rn First source general-purpose register (ZR is permitted, PC is not).

fcond The comparison condition to use. This is in the format of a standard Arm condition code. This parameter must be one of the following values:

- EQ.
- NE.
- CS.
- CC.
- MI.
- PL.
- VS.
- VC.
- HI.
- LS.
- GE.
- LT.
- GT.
- LE.

Restrictions

- Rd must not use the same register as SP and PC.
- Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Returns, in the destination register, the value of the first source register if the condition is TRUE, and otherwise returns the value of the second source register negated.

CSNEG example

```
// Conditional selection & negation of R2 & R3 based on two different conditions
MOV         R0, #4096
MOV         R1, #8192
CMP         R0, R1

CSNEG       R2, R1, R0, LE    // R2 = (cond == LE) ? R1 : ~R0
                              // In this case, the condition is TRUE,
                              // therefore, R2 = 8192
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
CSNEG       R3, R1, R0, GE    // R3 = (cond == GE) ? R1 : ~R0
                              // In this case, the condition is FALSE,
                              // therefore R3 = -4096
```

3.23.11 VCMP

Vector Compare.

Syntax

```
VCMP<v>.<dt> <fc>, Qn, Rm
VCMP<v>.<dt> <fc>, Qn, Qm
```

Parameters

Qm Second source vector register

Qn First source vector register

Rm Source general-purpose register (ZR is permitted, PC is not).

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - U8
  - U16
  - U32
  - S8
  - S16
  - S32
  - I8
  - I16
  - I32

fc The comparison condition to use. This parameter must be one of the following values:
- EQ
- NE
- CS
- HI
- GE
- LT
- GT
- LE

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use SP.

Post-conditions

There are no condition flags.

Operation

Perform a lane-wise comparison between each element in the first source vector register and either the respective elements in the second source vector register or the value of a general-purpose register. The resulting boolean conditions are placed in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

VCMP example

```
// 16-bit integer vector Comparison with scalar
// Saturate incrementing sequence to 8
MOV          R0, #0
MOV          R1, #8
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VIDUP.U16     Q0, R0, #2        // Incrementing sequence, starting at 0, increment of 2
VDUP.16       Q1,R1             // Q1[i] = 8   i={0..7}
VCMP.S16      GE, Q0, R1        // Q0[i] >= 8 ? (predicate state = 0xFF00)
                                // Q0 (in)  = [  0x0 0x2  0x4 0x6  0x8 0xA  0xC 0xE  ]
VPSEL         Q0, Q1, Q0        // Q0[i] = Q0[i] >= 8 ? Q1[i] : Q0[i] i={0..7}
                                // Q0 (out) = [  0x0 0x2  0x4 0x6  0x8 0x8  0x8 0x8  ]

// 2 x 16-bit integer vector Comparison
// Floor incrementing sequence to 2
VMOV.S16      Q1, #2            // Q1[i] = 2   i={0..7}
VCMP.S16      LE, Q0, Q1        // Q0[i] <= 2 ? (predicate state = 0x000F)
                                // Q0 (in)  = [  0x0 0x2  0x4 0x6  0x8 0x8  0x8 0x8  ]
VPSEL         Q0, Q1, Q0        // Q0[i] = Q0[i] <= 2 ? Q1[i] : Q0[i] i={0..7}
                                // Q0 (out) = [  0x2 0x2  0x4 0x6  0x8 0x8  0x8 0x8  ]

// misc alternating vector comparison variants
VCMP.S8       GE, Q0, R0        // Compare 8-bit integer vector with a scalar
VCMP.S16      GE, Q0, ZR        // Compare 16-bit integer vector with ZERO
VCMP.S32      GT, Q0, Q1        // Compare 32-bit integer vectors
```

3.23.12 VCMP (floating-point)

Vector Compare.

Syntax

```
VCMP<v>.<dt> <fc>, Qn, Rm
VCMP<v>.<dt> <fc>, Qn, Qm
```

Parameters

Qm Source vector register.

Qn First source vector register

Rm Source general-purpose register (ZR is permitted, PC is not).

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - `F32`
  - `F16`

fc The comparison condition to use. This parameter must be one of the following values:
- `EQ`
- `NE.`
- `GE.`
- `LT.`
- `GT.`
- `LE.`

v See Standard Assembler Syntax Fields

Restrictions

`fc` cannot be encoded as `01x`.

Post-conditions

There are no condition flags.

Operation

Perform a lane-wise comparison between each element in the first source vector register and either the respective elements in the second source vector register or the value of a general-purpose register. The resulting boolean conditions are placed in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

VCMP (floating-point) example

```
// 16-bit float Vector Comparison with scalar
// Saturate incrementing sequence to 8.0f16
MOV           R0, #0
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MOV           R1, #0x4800       // 8.0f16
VIDUP.U16     Q0, R0, #2        // Incrementing sequence, starting at 0, increment of 2
VCVT.F16.S16  Q0, Q0            // Convert into F16 vector
VDUP.16       Q1, R1            // Q1[i] = 8.0f16   i={0..7}
VCMP.F16      GE Q0, R1         // Q0[i] >= 8.0f16 ? (predicate state = 0xFF00)
                                // Q0 (in)  = [ 0.000 2.000  4.000 6.000  8.000 10.000
12.000 14.000 ]
VPSEL         Q0, Q1, Q0        // Q0[i] = Q0[i] >= 8.0f16 ? Q1[i] : Q0[i] i={0..7}
                                // Q0 (out) = [ 0.000  2.000   4.000  6.000   8.000  8.000
8.000  8.000  ]

// 2 x 16-bit float Vector Comparison
// Floor incrementing sequence to 2.0f16
MOV           R1, #0x4000       // 2.0f16
VDUP.16       Q1, Q1            // Q1[i] = 2.0f16   i={0..7}
VCMP.F16      LE, Q0, Q1        // Q0[i] <= 2.0f16 ? (predicate state = 0x000F)
                                // Q0 (in)  = [ 0.000  2.000   4.000  6.000   8.000  8.000
8.000  8.000  ]
VPSEL         Q0, Q1, Q0        // Q0[i] = Q0[i] <= 2.0f16 ? Q1[i] : Q0[i] i={0..7}
                                // Q0 (out) = [ 2.000 2.000  4.000 6.000  8.000 8.000  8.000
8.000 ]
```

3.23.13 VCTP

Create Vector Tail Predicate.

Syntax

```
VCTP<v>.<dt> Rn
```

Parameters

Rn The register containing the number of elements that need to be processed.

dt The size of the elements in the vector to process. This parameter must be one of the following values:

- 8
- 16
- 32
- 64

v See Standard Assembler Syntax Fields

Restrictions

Rn must not use SP.

Post-conditions

There are no condition flags.

Operation

Creates a predicate pattern in VPR.P0 such that any element numbered the value of Rn or greater is predicated. Any element numbered lower than the value of Rn is not predicated. If placed within a VPT block and a lane is predicated, the corresponding VPR.P0 pattern will also be predicated. The generated VPR.P0 pattern can be used by a subsequent predication instruction to apply tail predication on a vector register.

VCTP example

```
// Partial 16-bit vector load
MOV           R0, #5
VCTP.16       R0              // Enable 5 first 16-bit vector elements out of 8

VPST                          // Predicate activation (1 slot)

VLDRHT.16     Q0, [R1]        // 16-bit vector contiguous load of the
                              // 5 first elements (other vector elemements are zeroed)
                              // R1 points to 16-bit array containing {0, 1, 2, 3, 4, 5,
6, ...}
                              // Q0 = [ 0 1 2 3 4 0 0 0]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.23.14 VMAX, VMAXA

Vector Maximum, Vector Maximum Absolute.

Syntax

```
VMAXA<v>.<dt> Qda, Qm
VMAX<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register.

dt This parameter determines the following values:
- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Find the maximum value of the elements in the source operands, and store the result in the corresponding destination elements.

The absolute variant takes the elements from the destination vector, treating them as unsigned, and compares them to the absolute values of the corresponding elements in the source vector. The larger values are stored back into the destination vector.

VMAX example

```
// 16-bit integer Vector Maximum
MOV           R2, #0x0
VIDUP.U16     Q0,R2, #0x2          // 16-bit incrementing sequence starting at 0 with a step
of 2
VMOV.U16      Q1, #0x8             // 16-bit vector duplication
                                   // Q0 =[  0x0 0x2  0x4 0x6  0x8 0xA  0xC 0xE  ]
                                   // Q1 =[  0x8 0x8  0x8 0x8  0x8 0x8  0x8 0x8  ]

VMAX.U16      Q2, Q0, Q1         // Q2[i] = max(Q0[i], Q1[i])     i={0..7}
                                 // Q2 =[  0x8 0x8  0x8 0x8  0x8 0xA  0xC 0xE  ]
```

VMAXA example

```
// 16-bit integer Vector Maximum Absolute
MOV           R0, #0x0
MOV           R1, #-0x8
VIDUP.U16     Q0, R0, #0x2     // 16-bit incrementing sequence starting at 0 with a step of 2
VDUP.16       Q1, R1           // 16-bit vector duplication (-0x8 value)
                               // Q0 =[  0 2  4 6  8 A  C E  ]
                               // Q1 =[  -8 -8  -8 -8  -8 -8  -8 -8  ]

VMAXA.S16     Q0, Q1           // Q0[i] = max(|Q0[i]|, |Q1[i]|)     i={0..7}
                               // Q0 =[  8 8  8 8  8 A  C E  ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.23.15 VMAXNM, VMAXNMA (floating-point)

Vector Maximum, Vector Maximum Absolute.

Syntax

```
VMAXNM<v>.<dt> Qd, Qn, Qm
VMAXNMA<v>.<dt> Qda, Qm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Find the floating-point maximum number of the elements in the source operands, and store the result in the corresponding destination elements. It handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant takes the absolute values of the elements from the destination vector and compares them to the absolute values of the corresponding elements in the source vector. The larger values are stored back into the destination vector.

VMAXNM (floating-point) example

```
// 16-bit float Vector Maximum
MOV           R0, #0
MOV           R1, #8
VIDUP.U16     Q0, R0, #2     // 16-bit incrementing sequence starting at 0 with a step of 2
VCVT.F16.S16  Q0, Q0         // Convert into F16 vector
VDUP.16       Q1, R1         // 16-bit vector duplication
VCVT.F16.S16  Q1, Q1         // Convert into F16 vector
                             // Q0 = [ 0.0 2.0  4.0 6.0  8.0 10.0  12.0 14.0 ]
                             // Q1 = [ 8.0 8.0  8.0 8.0  8.0 8.0  8.0 8.0 ]

VMAXNM.F16    Q2, Q0, Q1     // Q2[i] = max(Q0[i], Q1[i])     i={0..7}
                             // Q2 = [ 8.0 8.0  8.0 8.0  8.0 10.0  12.0 14.0 ]
```

VMAXNMA (floating-point) example

```
// 16-bit float Vector Maximum Absolute
MOV           R0, #0
MOV           R1, #-8        // MVN R1, #7
VIDUP.U16     Q0, R0, #2     // 16-bit incrementing sequence starting at 0 with a step of 2
VCVT.F16.S16  Q0, Q0         // Convert into F16 vector
VDUP.16       Q1, R1         // 16-bit vector duplication (-8)
VCVT.F16.S16  Q1, Q1         // Convert into F16 vector
                             // Q0 = [  0.0 2.0  4.0 6.0  8.0 10.0  12.0 14.0  ]
                             // Q1 = [  -8.0 -8.0  -8.0 -8.0  -8.0 -8.0  -8.0 -8.0  ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMAXNMA.F16   Q0, Q1         // Q0[i] = max(|Q0[i]|, |Q1[i]|)     i={0..7}
                             // Q0 = [  8.0 8.0  8.0 8.0  8.0 10.0  12.0 14.0  ]
```

3.23.16 VMAXNMV, VMAXNMAV (floating-point)

Vector Maximum Across Vector, Vector Maximum Absolute Across Vector.

Syntax

```
VMAXNMAV<v>.<dt> Rda, Qm
VMAXNMV<v>.<dt> Rda, Qm
```

Parameters

Qm Source vector register.

Rda General-purpose source and destination register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

`Rda` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Find the maximum value of the elements in a vector register. Store the maximum value in the general-purpose destination register only if it is larger than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. For half-precision the upper half of the general-purpose register is cleared on writeback. This instruction handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant of the instruction compares the absolute value of vector elements.

VMAXNMV (floating-point) example

```
// Maximum accross 16-bit float vector
VLDRH.S16     Q0, [R0]          // 16-bit contiguous vector load
                                // Q0 = [ 0.0 2.0 4.0 6.0 8.0 -10.0 0.0 2.0 ]
MOV           R0, #0

VMAXNMV.F16   R0, Q0            // max(R0, Q0[i] i={0..7})
                                // R0 = 10.0F16
```

VMAXNMAV (floating-point) example

```
// Maximum Absolute accross 16-bit float vector
VLDRSH.S16     Q0, [R0]        // 16-bit contiguous vector load
                               // Q0 = [ 0.0 2.0 4.0 6.0 8.0 -10.0 0.0 2.0 ]
MOV            R0, #0

VMAXNMAV.F16   R0, Q0          // R0 = ABSMAX(R0, Q0[i] i={0..7})
                               // r0 = 10.0F16
```

3.23.17 VMAXV, VMAXAV

Vector Maximum Across Vector, Vector Maximum Absolute Across Vector.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VMAXV<v>.<dt> Rda, Qm
VMAXAV<v>.<dt> Rda, Qm
```

Parameters

Qm Source vector register.

Rda General-purpose source and destination register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rda must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Find the maximum value of the elements in a vector register. Store the maximum value in the general-purpose destination register only if it is larger than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. The result of the operation is sign-extended to 32 bits before being stored back.

The absolute variant of the instruction compares the absolute value of signed vector elements and treats the value in the general-purpose register as unsigned.

VMAXV example

```
// Maximum accross 16-bit integer vector
MOV          R0, #0            // Wrap sequence bottom
MOV          R1, #12           // Wrap sequence top
VIWDUP.U16   Q0, R0, R1, #2    // Alternating {0x0, 0x2 ,0x4, 0x6, 0x8, 0xA} 16-bit
sequence
MOV          R0, #0
                               // Q0 = [ 0x0  0x2  0x4  0x6  0x8  0xA  0x0  0x2 ]
VMAXV.S16    R0, Q0            // max(R0, Q0[i]   i={0..7})
                               // R0 = 0xA
```

VMAXAV example

```
// Maximum accross 16-bit integer vector
MOV          R0, #0            // Wrap sequence bottom
MOV          R1, #12           // Wrap sequence top
VIWDUP.U16   Q0, R0, R1, #2    // Alternating {0x0, 0x2 ,0x4, 0x6, 0x8, 0xA} 16-bit
sequence
MOV          R0, #0
                               // Q0 = [ 0x0  0x2  0x4  0x6  0x8  0xA  0x0  0x2 ]
VMAXV.S16    R0, Q0            // max(R0, Q0[i]   i={0..7})
                               // R0 = 0xA
```

3.23.18 VMIN, VMINA

Vector Minimum, Vector Minimum Absolute.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VMIN<v>.<dt> Qd, Qn, Qm
VMINA<v>.<dt> Qda, Qm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Find the minimum value of the elements in the source operands, and store the result in the corresponding destination elements.

The absolute variant takes the elements from the destination vector, treating them as unsigned, and compares them to the absolute values of the corresponding elements in the source vector. The smaller values are stored back into the destination vector.

VMIN, VMINA example

```
// 16-bit integer Vector Minimum
MOV          R2, #0
VIDUP.U16    Q0, R2, #2        // 16-bit incrementing sequence starting at 0 with a step
of 2
VMOV.U16     Q1, #8            // 16-bit vector duplication
                               // Q0 = [  0x0 0x2  0x4 0x6  0x8 0xA  0xC 0xE  ]
                               // Q1 = [  0x8 0x8  0x8 0x8  0x8 0x8  0x8 0x8  ]

VMIN.U16     Q2, Q0, Q1        // Q2[i] = min(Q0[i], Q1[i])     i={0..7}
                               // Q2 = [  0x0 0x2  0x4 0x6  0x8 0x8  0x8 0x8  ]
```

VMIN, VMINA example

```
// 16-bit integer Vector Minimum Absolute
MOV          R0, #0
MOV          R1, #-8
VIDUP.u16    Q0, R0, #2     // 16-bit incrementing sequence starting at 0 with a step of 2
VDUP.16      Q1, R1         // 16-bit vector duplication (-8 value)
                            // Q0 = [  0x0 0x2  0x4 0x6  0x8 0xA  0xC 0xE  ]
                            // Q1 = [  -0x8 -0x8  -0x8 -0x8  -0x8 -0x8  -0x8 -0x8  ]

VMINA.S16    Q0, Q1         // Q0[i] = min(|Q0[i]|, |Q1[i]|)     i={0..7}
                            // Q0 = [  0x0 0x2  0x4 0x6  0x8 0x8  0x8 0x8  ]
```

3.23.19 VMINNM, VMINNMA (floating-point)

Vector Minimum, Vector Minimum Absolute.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VMINNM<v>.<dt> Qd, Qn, Qm
VMINNMA<v>.<dt> Qda, Qm
```

Parameters

Qd Destination vector register.

Qda Source and destination vector register.

Qm Source vector register.

Qn Source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Find the floating-point minimum number of the elements in the source operands, and store the result in the corresponding destination elements. It handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant takes the absolute values of the elements from the destination vector and compares them to the absolute values of the corresponding elements in the source vector. The smaller values are stored back into the destination vector.

VMINNM, VMINNMA (floating-point) example

```
// 16-bit float Vector Minimum
MOV           R0, #0
MOV           R1, #8
VIDUP.U16     Q0, R0, #2     // 16-bit incrementing sequence starting at 0 with a step of 2
VCVT.F16.S16  Q0, Q0         // Convert into F16 vector
VDUP.16       Q1, R1         // 16-bit vector duplication
VCVT.F16.S16  Q1, Q1         // Convert into F16 vector

VMINNM.F16    Q2, Q0, Q1     // Q2[i] = min(Q0[i], Q1[i])     i={0..7}
                             // Q0 = [ 0.0 2.0  4.0 6.0  8.0 10.0  12.0 14.0 ]
                             // Q1 = [ 8.0 8.0  8.0 8.0  8.0 8.0  8.0 8.0 ]
                             // Q2 = [ 0.0 2.0  4.0 6.0  8.0 8.0  8.0 8.0 ]
```

VMINNM, VMINNMA (floating-point) example

```
// 16-bit float Vector Minimum Absolute
MOV           R0, #0
MOV           R1, #-8        // MVN R1, #7
VIDUP.U16     Q0, R0, #2     // 16-bit incrementing sequence starting at 0 with a step of 2
VCVT.F16.S16  Q0, Q0         // Convert into F16 vector
VDUP.16       Q1, R1         // 16-bit vector duplication (-8)
VCVT.F16.S16  Q1, Q1         // Convert into F16 vector
                             // Q0 = [  0.0 2.0  4.0 6.0  8.0 10.0  12.0 14.0  ] (before
VMINNMA)
                             // Q1 = [ -8.0 -8.0  -8.0 -8.0  -8.0 -8.0  -8.0 -8.0  ]

VMINNMA.F16   Q0, Q1         // Q0[i] = min(|Q0[i]|, |Q1[i]|)     i={0..7}
                             // Q0 = [ 0.0 2.0  4.0 6.0  8.0 8.0  8.0 8.0 ] (after VMINNMA)
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.23.20 VMINNMV, VMINNMAV (floating-point)

Vector Minimum Across Vector, Vector Minimum Absolute Across Vector.

Syntax

```
VMINNMAV<v>.<dt> Rda, Qm
VMINNMV<v>.<dt> Rda, Qm
```

Parameters

Qm Source vector register.

Rda General-purpose source and destination register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

`Rda` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Find the minimum value of the elements in a vector register. Store the minimum value in the general-purpose destination register only if it is smaller than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. For half-precision the upper half of the general-purpose register is cleared on writeback. This instruction handles NaNs in consistence with the IEEE754-2008 specification, and returns the numerical operand when one operand is numerical and the other is a quiet NaN.

The absolute variant of the instruction compares the absolute value of vector elements.

VMINNMAV (floating-point) example

```
// Minimum Absolute accross 16-bit float vector
VLDRH.16      Q0, [R0]          // 16-bit contiguous vector load
MOV           R0, #100.0
                                // Q0 = [  0.0 2.0  4.0 6.0  8.0 -10.0 0.0 2.0  ] (before
VMINNMAV)

VMINNMAV.F16  R0, Q0            // R0 = absmin(R0, Q0[i] i={0..7})
                                // R0 = 0.0F16
```

VMINNMV (floating-point) example

```
// Minimum accross 16-bit float vector
VLDRH.16      Q0, [R0]          // 16-bit vector contiguous load
MOV           R0, #100.0
                                // q0 = [  0.0 2.0  4.0 6.0  8.0 -10.0 0.0 2.0  ] (before
VMINNMV)

VMINNMV.F16   R0, Q0            // min(R0, Q0[i] i={0..7})
                                // R0 = -10.0F16
```

3.23.21 VMINV, VMINAV

Vector Minimum Across Vector, Vector Minimum Absolute Across Vector.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VMINAV<v>.<dt> Rda, Qm
VMINV<v>.<dt> Rda, Qm
```

Parameters

Qm Source vector register.

Rda General-purpose source and destination register.

dt This parameter determines the following values:
- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rda must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Find the minimum value of the elements in a vector register. Store the minimum value in the general-purpose destination register only if it is smaller than the starting value of the general-purpose destination register. The general-purpose register is read as the same width as the vector elements. The result of the operation is sign-extended to 32 bits before being stored back.

The absolute variant of the instruction compares the absolute value of signed vector elements and treats the value in the general-purpose register as unsigned.

VMINV example

```
// Minimum accross 16-bit integer vector
VLDRH.16      Q0, [R0]          // 16-bit vector contiguous load
MOV           R0, #0
                                // Q0 = [ 0x0 0x2 0x4 0x6 0x8 -0xA 0x0 0x2 ]

VMINV.S16     R0, Q0            // min(R0, Q0[i] i={0..7}]
                                // R0 = -10
```

VMINAV example

```
// Minimum Absolute accross 16-bit integer vector
MOV           R2, #0
MOV           R0, #0             // Wrap sequence bottom
MOV           R1, #0xC           // Wrap sequence top
VIWDUP.U16    Q0, R0, R1, #2     // Alternating {0x0, 0x2 ,0x4, 0x6, 0x8, 0xA} 16-bit
sequence
                                 // Q0 = [ 0x0  0x2  0x4  0x6  0x8  0xA  0x0  0x2 ]
MOV           R0, #0x64

VMINAV.S16    R0, Q0            // absmin(R0, Q0[i] i ={0..7})
                                // R0 = 0
```

3.23.22 VPNOT

Vector Predicate NOT.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VPNOT<v>
```

Parameters

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Inverts the predicate condition in VPR.P0. The VPR.P0 flags for predicated lanes are zeroed.

VPNOT example

```
// Vector Predicate NOT. Inverts the predicate condition in VPR.P0
// 32-bit vector selection based on inverted condition
MOV             R0, #0
VIDUP.U32       Q0, R0, #1   // 32-bit Generator, increment step of 1
VDDUP.U32       Q1, R0, #1   // 32-bit Generator, decrement step of 1
VMOV.S32        Q2, #10      // Q2[i] = 10        i={0..3}
VMOV.S32        Q3, #20      // Q3[i] = 20        i={0..3}
VCMP.S32        LE, Q0, Q1   // Q0[i] <= Q1[i] ?  i={0..3}, set VPR.P0
                             // Q0 = [ 0 1 2 3 ]
                             // Q1 = [ 4 3 2 1 ]

VPNOT                        // Invert current predicate conditions VPR.P0
VPSEL           Q2, Q2, Q3   // Performs selection based on inverted condition
                             // Q2[i] = Q0[i] > Q1[i] ? Q2[i] : Q3[i]  i={0..3}
                             // Q2 = [ 20 20 20 10 ]
```

3.23.23 VPST

Vector Predicate Set Then.

Syntax

```
VPST{x{y{z}}}
```

Parameters

x Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into `fcond[0]`, a `1` in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted

y Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

z Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Predicates the following instructions, up to a maximum of four instructions. This instruction is similar to VPT. However no comparison is performed and instead the current value of VPR.P0 is used as the predicate condition.

VPST example

```
// Partial 16-bit vector load
MOV          R0, #5
VCTP.16      R0              // Enable 5 first 16-bit vector elements out of 8,
VPR=0x000003FF
VPST                         // Predicate activation (1 slot, VPR=0x008803FF)

VLDRHT.16    Q0, [R1]        // 16-bit vector contiguous load of the 5 first elements
(other vector elemements are zeroed)
                             // R1 points to 16-bit array containing {0, 1, 2, 3, 4, 5,
6, ...}
                             // Q0 = [ 0 1 2 3 4 0 0 0]

// partial 32-bit vector negation / multiplication
MOVS           R1, 0xFF      // 2 first 32-bit lanes mask
MOVS           R0, #0
MOV            R2, #1000
VIDUP.U32      Q0, R0, #1    // 32-bit Generator, increment step of 1
VMSR           P0, R1        // Set predicate, VPR=0x000000FF
                             // Q0 = [ 0 1 2 3 ]
VPSTT                        // Activate predication for the 2 next slots, VPR=0x004400FF

VNEGT.S32      Q0, Q0        // Negate active lanes
VMULT.S32      Q0, Q0, R2    // Multiply active lanes
                             // Q0 = [ 0 -1000 2 3 ]

// Partial 32-bit vector mutiply based on comparison
MOVS           R0, #0
MOVS           R1, #2
MOV            R2, #1000
VIDUP.U32      Q0, R0, #1    // 32-bit Generator, increment step of 1
                             // Q0 = [ 0 1 2 3 ]
VCMP.S32       EQ, Q0, R1    // Compare Q2[i]  to Rr2, VPR=0x00000F00

VPSTTT                       // Activate predication for the 3 next slots, VPR=0x00220F00
VMULT.S32      Q0, Q0, R2    // Q0[i] = Q0[i] * R2 for active lane
                             // Q0 = [ 0 1 2000 3 ]
VMULT.S32      Q0, Q0, R2    // Q0[i] = Q0[i] * R2 for active lane
                             // Q0 = [ 0 1 2000000 3 ]
VMULT.S32      Q0, Q0, R2    // Q0[i] = Q0[i] * R2 for active lane
                             // Q0 = [ 0 1 2000000000 3 ]
```

3.23.24 VPT

Vector Predicate Then.

Syntax

```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

Parameters

Qm Source vector register.

Qn Source vector register.

Rm Source general-purpose register (ZR is permitted, PC is not). ZR is the zero register and behaves as RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt Indicates the size of the elements in the vector. This parameter must be one of the following values

- `U8`
- `U16`
- `U32`
- `S8`
- `S16`
- `S32`
- `I8`
- `I16`
- `I32`

fc The comparison condition to use. This parameter must be one of the following values:

- `EQ`.
- `NE`.
- `CS`.
- `HI`.
- `GE`.
- `LT`.
- `GT`.
- `LE`.

x Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into fcond[0], a 1 in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted

y Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

z Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

Restrictions

`Rm` must not use `SP`

Post-conditions

There are no condition flags.

Operation

Predicates the following instructions, up to a maximum of four instructions, by masking the operation of instructions on a per-lane basis based on the VPR.P0 predicate values. The predicated instructions are referred to as the `Vector Predication Block` or simply the `VPT Block`. The VPR.P0 predicate values may be inverted after each instruction in the VPT block based on the mask fields (see x, y, and z).

VPT example

```
// Vector Predicate Then (Integer).
// 32-bit integer vector capping
MOVS            R0, #1
MOVS            R1, #3
VIDUP.U32       Q0, R0, #1  // 32-bit Generator, increment step of 1
                            // Q0 = [ 1 2 3 4]

VPTE.S32        GE, Q0, R1  // Enable lanes greater or equal than R1, VPR=0x00CCFF00
VDUPT.32        Q0, R1      // Set Q0[i] to r1 for active lanes
                            // Q0 = [ 1 2 3 3 ]
VMULE.S32       Q0, q0, R1  // Q0[i] = Q0[i] * R1 for complement lanes
                            // Q0 = [ 3 6 3 3 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.23.25 VPT (floating-point)

Vector Predicate Then.

Syntax

```
VPT{x{y{z}}}.<dt> <fc>, Qn, Rm
VPT{x{y{z}}}.<dt> <fc>, Qn, Qm
```

Parameters

`Qm` Source vector register.

`Qn` Source vector register.

`Rm` Source general-purpose register (ZR is permitted, PC is not).

`dt` Indicates the floating-point format used. This parameter must be one of the following values

- `F32`
- `F16`

`fc` The comparison condition to use. This parameter must be one of the following values:

- `EQ.`
- `NE.`
- `GE.`
- `LT.`
- `GT.`
- `LE.`

`x` Specifies the condition for an optional second instruction in the VPT block, and whether the condition is the same as for the first instruction (T) or its inverse (E). This is encoded in the mask field in a similar way to the IT instruction, except that rather than encoding T and E directly into `fcond[0]`, a `1` in the corresponding mask bit indicates that the previous predicate value in VPR.P0 should be inverted

`y` Specifies the condition for an optional third instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

`z` Specifies the condition for an optional fourth instruction in the VPT block. It is encoded in the mask field in the same way as the x field.

Restrictions

`FC` must not use the same register as `01x`

Post-conditions

There are no condition flags.

Operation

Predicates the following instructions, up to a maximum of four instructions, by masking the operation of instructions on a per-lane basis based on the VPR.P0 predicate values. The predicated instructions are referred to as the `Vector Predication Block` or simply the `VPT Block`. The VPR.P0 predicate values may be inverted after each instruction in the VPT block based on the mask fields (see x, y, and z).

VPT (floating-point) example

```
// Vector Predicate Then (Float)
// 32-bit floating vector capping
MOVS            R0, #1
MOVS            R1, #3
MOVW            R2, #0
MOVT            R2, #0x4040  // R2 = 3.0f

VIDUP.U32       Q0, R0, #1   // 32-bit Generator, increment step of 1
VCVT.F32.S32    Q0,  Q0      // Convert to float
                             // Q0 = [ 1.0 2.0 3.0 4.0 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VPTE.F32        GE, Q0, R2   // Enable lanes greater or equal than R2, VPR=0x00CCFF00
VDUPT.32        Q0, R2       // Set Q0[i] to R2 for active lanes
                             // Q0 = [ 1.0 2.0 3.0 3.0 ]
VMULE.F32       Q0, Q0, R2   // Q0[i] = Q0[i] * R1 for complement lanes
                             // Q0 = [ 3.0 6.0 3.0 3.0 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.24 Arm®v8.1-M vector load and store operations instructions

Reference material for the Cortex-M55 processor Armv8.1-M vector load and store operations instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.24.1 List of Arm®v8.1-M vector load and store operations instructions

An alphabetically ordered list of the Armv8.1-M vector load and store operations instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-18 Armv8.1-M vector load and store operations instructions

Mnemonic	Brief description	See
VLD2	Vector Deinterleaving Load - Stride 2	*3.24.2 VLD2* on page 3-325
VLD4	Vector Deinterleaving Load - Stride 4	*3.24.3 VLD4* on page 3-326
VLDR	VLDR (System Register) Load System Register	*3.20.13 VLDR* on page 3-233
VLDRB, VLDRH, VLDRW	Vector Load Register	*3.24.5 VLDRB, VLDRH, VLDRW* on page 3-329
VLDRB, VLDRH, VLDRW, VLDRD	VLDRB, VLDRH, VLDRW, VLDRD (vector) Vector Gather Load	*3.24.6 VLDRB, VLDRH, VLDRW, VLDRD (vector)* on page 3-330
VST2	Vector Interleaving Store - Stride 2	*3.24.7 VST2* on page 3-332
VST4	Vector Interleaving Store - Stride 4	*3.24.8 VST4* on page 3-333
VSTR	VSTR (System Register) Store System Register	*3.20.33 VSTR* on page 3-253
VSTRB, VSTRH, VSTRW	Vector Store Register	*3.24.10 VSTRB, VSTRH, VSTRW* on page 3-335
VSTRB, VSTRH, VSTRW, VSTRD	VSTRB, VSTRH, VSTRW, VSTRD (vector) Vector Scatter Store	*3.24.11 VSTRB, VSTRH, VSTRW, VSTRD (vector)* on page 3-336

3.24.2 VLD2

Vector Deinterleaving Load - Stride 2.

Syntax

```
VLD2<pat>.<size> {Qd, Qd+1}, [Rn]
VLD2<pat>.<size> {Qd, Qd+1}, [Rn]!
```

Parameters

Qd Destination vector register.

Rn The base register for the target address.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pat Specifies the pattern of register elements and memory addresses to access. This parameter must be one of the following values:
- `0`
- `1`

size
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - `8`
  - `16`
  - `32`

Restrictions

- `Rn` must not use `PC`.
- `Rn` must not use `sp` when the writeback variant is used.
- `Qd` must only access registers numbers lower than `R6`.

Post-conditions

There are no condition flags.

Operation

Loads two 64-bit contiguous blocks of data from memory and writes them to parts of 2 destination registers. The parts of the destination registers written to, and the offsets from the base address register, are determined by the `pat` parameter. If the instruction is executed two times with the same base address and destination registers, but with different `pat` values, the effect is to load data from memory and to deinterleave it into the specified registers with a stride of 2. The base address register can optionally be incremented by 32.

VLD2 example

```
// Single-precision floating-point stereo stream gain control
MOV      R1,#0x6666 // stereo gain, right = 0.9F
MOVT     R1,#0x3F66

MOV      R2,#0xCCCD// stereo gain, left = 0.8F
MOVT     R2,#0x3F4C

VLD20.32       {Q0, Q1}, [R0]  // Load interleaved-2 part 1, base addr = R0
VLD21.32       {Q0, Q1}, [R0]  // Load interleaved-2 part 2, base addr = R0
                               // R0, points to array containing {0, 0.1, 0.2, 0.3...}
                               // At this stage, Q0 contains even 32-bit memory sample.
                               // For example, the right channel)
                               // At this stage, Q1 contains odd 32-bit memory samples.
                               // For example, the left channel)
                               // Q0 (in) = [ 0.000 0.200 0.400 0.600 ]%Right channel in
                               // Q1 (in) = [ 0.100 0.300 0.500 0.700 ]%Left channel in
VMUL.F32       Q0, Q0, R1      // Apply gain, right channel, Q0[i] = Q0[i] * 0.9
VMUL.F32       Q1, Q1, R2      // Apply gain, right channel, Q1[i] = Q1[i] * 0.8
                               // Q0 (out) = [ 0.000 0.144 0.288 0.432 ]%Right channel out
                               // Q1 (out) = [ 0.08 0.24 0.4 0.56 ]%Left channel out
```

3.24.3 VLD4

Vector Deinterleaving Load - Stride 4.

Syntax

```
VLD4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]!
VLD4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]
```

Parameters

Qd Destination vector register.

Rn The base register for the target address.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

pat Specifies the pattern of register elements and memory addresses to access. This parameter must be one of the following values:
- `0`
- `1`
- `2`
- `3`

size
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - `8`
  - `16`
  - `32`

Restrictions

- `Rn` must not use `PC`.
- `Rn` must not use `SP` when the writeback variant is used.
- `Qd` must only access registers numbers lower than `R4`.

Post-conditions

There are no condition flags.

Operation

Loads two 64-bit contiguous blocks of data from memory and writes them to parts of 4 destination registers. The parts of the destination registers written to, and the offsets from the base address register, are determined by the `pat` parameter. If the instruction is executed four times with the same base address and destination registers, but with different `pat` values, the effect is to load data from memory and to deinterleave it into the specified registers with a stride of 4. The base address register can optionally be incremented by 64.

VLD4 example

```
// 4 x 4 32-bit matrix transpose
// R0 points to [ 0.0, 0.1, 0.2, 0.3, 0.4...1.4, 1.5]
VLD40.32            {Q0, Q1, Q2, Q3}, [R0]     // Load interleaved-4 part 1
VLD41.32            {Q0, Q1, Q2, Q3}, [R0]     // Load interleaved-4 part 2
VLD42.32            {Q0, Q1, Q2, Q3}, [R0]     // Load interleaved-4 part 3
VLD43.32            {Q0, Q1, Q2, Q3}, [R0]     // Load interleaved-4 part 4

VSTRW.32            Q0, [R1, #0]               // Contiguous store Q0 at destination
                                               //(1st row)
VSTRW.32            Q1, [R1, #16]              // Contiguous store Q1 at destination
                                               //(2nd row)
VSTRW.32            Q2, [R1, #32]              // Contiguous store Q2 at destination
                                               //(3rd row)
VSTRW.32            Q3, [R1, #48]              // Contiguous store Q3 at destination
                                               //(4th row)

                                                // R1 memory contains
                                                // =[
                                                // 0.0, 0.400000, 0.800000, 1.200000,...
                                                // 0.100000, 0.500000, 0.900000, 1.300000,...
                                                // 0.200000, 0.600000, 1.0, 1.400000,...
                                                // 0.300000, 0.700000, 1.100000, 1.500000,...
                                                // ];
```

3.24.4 VLDR (System Register)

Load System Register.

Syntax

```
VLDR<c> <reg>, [Rn{, #+/-<imm>}]
VLDR<c> <reg>, [Rn, #+/-<imm>]!
VLDR<c> <reg>, [Rn], #+/-<imm>
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

`Rn` The base register for the target address.

`c` See Standard Assembler Syntax Fields

`imm` The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

`reg` The system register to access This parameter must be one of the following values:
- FPSCR.
- FPSCR_nzcvqc.
- VPR.
- P0.
- FPCXT_NS.
- FPCXT_S.

Restrictions

- `reg` must not use `R0`
- `reg` must not use the same register as `R4` to `R7`
- `reg` must not use `R3`
- `Rn` must not use `PC`
- `reg` must not use the same register as `R8` to `R11`
- `Rn` must not use `SP` when the writeback variant is used

Post-conditions

There are no condition flags.

Operation

Load a system register from memory. The target address is calculated from a base register plus an immediate offset.

Access to the FPCXT payloads generates an UNDEFINED exception if the instruction is executed from Non-secure state.

If CP10 is not enabled and either the Main extension is not implemented or the Floating-point context is active, access to FPCXT_NS will generate a NOCP UsageFault.

Accesses to FPCXT_NS will not trigger lazy state preservation if there is no active Floating-point context.

Accesses to FPCXT_NS do not trigger Floating-point context creation regardless of the value of FPCCR.ASprocessorN.

FPSCR_nzcvqc allows access to FPSCR condition and saturation flags.

The VPR register can only be accessed from privileged mode, otherwise the instruction behaves as a NOP.

FPCXT_NS, enables saving and restoration of the Non-secure floating-point context.

If the Floating-point extension and MVE are implemented and Floating-point context is active then the current FPSCR value is accessed, otherwise the instruction behaves as a NOP.

FPCXT_S, enables saving and restoration of the Secure floating-point context.

VLDR (System Register) example

```
VLDR           P0,[SP,#0]          // Set VPR.p0 by loading from stack
VPST                               // Predicate activation
VDUPT.32       Q0,R0               // Set Q0[i] to R0 for active lanes

VLDR           VPR,[SP]!           // Set VPR by loading from stack,
                                   // Post incremented load
VLDR           FPSCR,[SP,#4]       // Set FPSCR by loading from stack
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VLDR           FPSCR_NZCVQC,[SP,#8] // Update the N, Z, C, V, and QC flags by
                                    // Loading FPSCR from stack
```

3.24.5 VLDRB, VLDRH, VLDRW

Vector Load Register.

Syntax

```
VLDR{B,H,W}<v>.<dt> Qd, [Rn{, #+/-<imm>}]
VLDR{B,H,W}<v>.<dt> Qd, [Rn, #+/-<imm>]!
VLDR{B,H,W}<v>.<dt> Qd, [Rn], #+/-<imm>
```

Parameters

Qd Destination vector register.

Rn The base register for the target address.

dt This parameter determines the following values:
- S16
- U16
- S32
- U32

imm The signed immediate value that is added to base register to calculate the target address. No restrictions for the VLDRB variants. This value must be a multiple of 2 for the VLDRH variants. This value must be a multiple of 4 for the VLDRW variants.

v See Standard Assembler Syntax Fields

Restrictions

- Rn must not use PC
- Rn must not use SP when the writeback variant is used

Post-conditions

There are no condition flags.

Operation

Load consecutive elements from memory into a destination vector register. Each element loaded will be the zero or sign-extended representation of the value in memory. In indexed mode, the target address is calculated from a base register offset by an immediate value. Otherwise, the base register address is used directly. The sum of the base register and the immediate value can optionally be written back to the base register. Predicated lanes are zeroed instead of retaining their previous values.

VLDRB example

```
// R0, points to byte array containing {0, 1, 2, 3, 4...}
VLDRB.S8      Q0, [R0, #8]       // Contiguous load, pre-index=8
                                 // Q0 = [  8 9 10 11  12 13 14 15  16 17 18 19  20 21 22
23 ]
VLDRB.S16     Q0, [R0, #0]       // Contiguous load + widening, load bytes into 16-bit vector
                                 // Q0 = [  0 1  2 3  4 5  6 7  ]
VLDRB.S32     Q0, [R0, #0]       // Contiguous load + widening, load bytes into 32-bit vector
                                 // Q0 = [ 0 1 2 3 ]
VLDRB.S8      Q0, [R0, #8]!      // Contiguous load, pre-index by 8 + write-back
                                 // Q0 = [  8 9 10 11  12 13 14 15  16 17 18 19  20 21 22
23 ]
                                 // R0 = R0 + 8
VLDRB.S8      Q0, [R0], #8       // Contiguous load, post-increment by 8
                                 // Q0 = [  8 9 10 11  12 13 14 15  16 17 18 19  20 21 22
23 ]
                                 // R0 = R0 + 8
```

VLDRH example

```
// R1, points to short array containing {0, 1, 2, 3, 4...}
VLDRH.S16     Q0, [R1, #8]       // Contiguous load, pre-index=8 (= 4 half words offset)
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                   // Q0 = [  4 5  6 7  8 9  10 11  ]
VLDRH.S32     Q0, [R1]             // Contiguous load + widening, load half words into
                                   //32-bit vector
                                   // Q0 = [ 0 1 2 3 ]
VLDRH.S16     Q0, [R1, #8]!        // Contiguous load, pre-index by 8 + write-back
                                   // Q0 = [  4 5  6 7  8 9  10 11  ]
                                   // R1 = R1 + 8
VLDRH.S16     Q0, [R1], 16         // Contiguous load, post-increment by 16
                                   // Q0 = [  4 5  6 7  8 9  10 11  ]
                                   // R1 = R1 + 16
```

VLDRW example

```
// R2, points to long array containing {0, 1, 2, 3, 4...}
VLDRW.S32     Q0, [R2, #8]         // Contiguous load, pre-index=8 (= 2 words offset)
                                   // Q0 = [ 2 3 4 5]
VLDRW.S32     Q0, [R2, #16]!       // Contiguous load, pre-index by 8 (= 4 words offset)
                                   //+ write-back
                                   // Q0 = [ 4 5 6 7]
VLDRW.S32     Q0, [R2], #16        // Contiguous load, post-increment by 16
                                   // Q0 = [ 2 3 4 5]
```

3.24.6 VLDRB, VLDRH, VLDRW, VLDRD (vector)

Vector Gather Load.

Syntax

```
VLDRB<v>.<dt> Qd, [Rn, Qm]
VLDR{H,W,D}<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
VLDR{W,D}<v>.<dt> Qd, [Qm{, #+/-<imm>}]
VLDR{W,D}<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

Parameters

Qd Destination vector register.

Qm The base register for the target address. Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

Rn The base register for the target address.

dt Unsigned flag:

- S indicates signed.
- U indicates unsigned. operations that do not perform widening are always unsigned.
- The equivalent sized floating and signless datatypes are allowed but are an alias for the unsigned version. this parameter must be one of the following values
  - `U8`
  - `S16`
  - `U16`
  - `S32`
  - `U32`

imm The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4, for the VLDRW variant. This value must be a multiple of 8, for the VLDRD variant.

os The amount by which the vector offset is left shifted by before being added to the general-purpose base address. If the value is present it must correspond to memory transfer size (1=half word, 2=word, 3=double word). This parameter must be one of the following values:

- `omitted`
- `offset scaled`

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

- `Rn` must not use `PC`
- `Rn` must not use `SP` when the writeback variant is used
- `Qd` must not use the same register as `Qm`

Post-conditions

There are no condition flags.

Operation

Load a byte, halfword, word, or doubleword from memory at the address contained in either of the following:

- A base register R[n] plus an offset contained in each element of Q[m], optionally shifted by the element size.
- Each element of Q[m] plus an immediate offset. The base element can optionally be written back, irrespective of predication, with that value incremented by the immediate or by the immediate scaled by the memory element size.

Each element loaded will be the zero or sign-extended representation of the value in memory. The result is written back into the corresponding element in the destination vector register Q[d]. Predicated lanes are zeroed instead of retaining their previous values.

VLDRB(vector) example

```
MOV          R2, #1
VIDUP.U8     Q1,R2,#1          // Generates incrementing sequence,
                               // starting at 1 with increments of 1
VLDRB.S8     Q0, [R0, Q1]      // Byte gather load, base in R0, indexes in Q1

MOV          R2, #1
VIDUP.U16    Q1,R2,#1          // Generates incrementing sequence,
                               // starting at 1 with increments of 1
VLDRB.S16    Q0, [R0, Q1]      // Byte gather load with widening, base in R0, indexes in Q1
```

VLDRH (vector) example

```
MOV          R2, #0
VIDUP.U16    Q1,R2,#1             // Generates incrementing sequence,
                                  // starting at 0 with increments of 1
VLDRH.S16    Q0, [R0, Q1, UXTW #1] // Halfword gather load, base in r0, indexes in q1,
scale applied

MOV          R2, #0
VIDUP.U32    Q1,R2,#1             // Generates incrementing sequence,
                                  // starting at 0 with increments of 1
VLDRH.S32    Q0, [R0, Q1, UXTW #1] // Halfword gather load with widening,
                                  // base in R0, indexes in Q1, scale applied
```

VLDRW (vector) example

```
MOV          R2, #0
VIDUP.U32    Q1,R2,#1             // Generates incrementing sequence,
                                  // starting at 0 with increments of 1
VLDRW.S32    Q0, [R0, Q1, UXTW #2] // Word gather load, base in R0, indexes in Q1, scale
applied

MOV          R2, #0
VIDUP.U32    Q1,R2,#8             // Generates incrementing sequence,
                                  // starting at 0 with increments of 8
VADD.S32     Q1, Q1, R0           // R0 contains base address,
                                  // Q1 = vector of addresses = [R0 + 0, R0 + 8, R0 + 16,
R0 + 24]
VLDRW.S32    Q0, [Q1, #0]         // Word gather load, base in Q1, pre-index=0

MOV          R2, #0
VIDUP.U32    Q1,R2,#8             // Generates incrementing sequence,
                                  // starting at 0 with increments of 8
VADD.S32     Q1, Q1, R0           // R0 contains base address,
                                  // Q1 = vector of addresses = [R0 + 0, R0 + 8, R0 + 16,
R0 + 24]
VLDRW.S32    Q0, [Q1, #16]!       // Word gather load, base in Q1, pre-index=16,
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    // write-back applied
                                    // Load [R0 + 16, R0 + 24, R0 + 32, R0 + 40]
```

VLDRD (vector) example

```
MOVS        R0, #1                  // Build Q1 = {1LL, 2LL}
MOVS        R2, #2
VMOV.S32    Q1, #0
VMOV        Q1[2], Q1[0], R0, R2
VLDRD.S64   Q0, [R0, Q1, UXTW #3]  // Double word gather load, base in R0,
                                    // indexes in Q1, scale applied
VLDRD.S64   Q0, [Q1, #16]          // Double word gather load, base in Q1, pre-index=16
VLDRD.S64   Q0, [Q1, #64]!         // Double word gather load, base in Q1, pre-index=64,
                                    // write-back applied
```

3.24.7 VST2

Vector Interleaving Store - Stride 2.

Syntax

```
VST2<pat>.<size> {Qd, Qd+1}, [Rn]
VST2<pat>.<size> {Qd, Qd+1}, [Rn]!
```

Parameters

Qd Source vector register.

Rn The base register for the target address.

pat Specifies the pattern of register elements and memory addresses to access. This parameter must be one of the following values:

- 0
- 1

size Indicates the size of the elements in the vector. This parameter must be one of the following values:

- 8
- 16
- 32

Restrictions

- Rn must not use PC
- Rn must not use Sp when the writeback variant is used
- Qd must only access registers numbers lower than R6

Post-conditions

There are no condition flags.

Operation

Saves two 64-bit contiguous blocks of data to memory made up of multiple parts of 2 source registers. The parts of the source registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 2 times with the same base address and source registers, but with different pat values, the effect is to interleave the specified registers with a stride of 2 and to save the data to memory. The base address register can optionally be incremented by 32.

VST2 example

```
// Vector Interleaving Store - Stride 2
// Complex conjugate
MOV         R2, #3                // 3 loop iterations
VMOV.S32    Q2, #0                // Clear Q2
WLS         LR, R2, 1F            // Low overhead while loop start
2:
VLD20.32    {Q0,Q1}, [R0]         // Load + 32-bit stride 2 de-interleaving part 1
VLD21.32    {Q0,Q1}, [R0]!        // Load + 32-bit stride 2 de-interleaving part 2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                 // (Q0 contains real numbers, Q1 contains imaginary
numbers)
VQSUB.S32      Q1, Q2, Q1        // Q1[i] = SSAT32( -Q1[i]), i={0..3}
VST20.32       {Q0,Q1}, [R1]     // Store + 32-bit stride 2 interleaving part 1
VST21.32       {Q0,Q1}, [R1]!    // Store + 32-bit stride 2 interleaving part 2
LE             LR, 2B            // While loop end
1:
```

3.24.8 VST4

Vector Interleaving Store - Stride 4.

Syntax

```
VST4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]
VST4<pat>.<size> {Qd, Qd+1, Qd+2, Qd+3}, [Rn]!
```

Parameters

Qd Source vector register.

Rn The base register for the target address.

pat Specifies the pattern of register elements and memory addresses to access. This parameter must be one of the following values:

- 0
- 1
- 2
- 3

size Indicates the size of the elements in the vector. This parameter must be one of the following values:

- 8
- 16
- 32

Restrictions

- Rn must not use PC
- Rn must not use SP when the writeback variant is used
- Qd must only access registers numbers lower than R4

Post-conditions

There are no condition flags.

Operation

Saves two 64-bit contiguous blocks of data to memory made up of multiple parts of 4 source registers. The parts of the source registers written to, and the offsets from the base address register, are determined by the pat parameter. If the instruction is executed 4 times with the same base address and source registers, but with different pat values, the effect is to interleave the specified registers with a stride of 4 and to save the data to memory. The base address register can optionally be incremented by 64.

VST4 example

```
// Vector Interleaving Store - Stride 4.
// 4x4 32-bit float matrix transpose
// R0, points to array containing {
//     0.0, 0.1, 0.2, 0.3,...
//     0.4, 0.5, 0.6, 0.7,...
//     0.8, 0.9, 1.0, 1.1,...
//     1.2, 1.3, 1.4, 1.5}

VLDRW.32            Q0, [R0, #(0*16)]         // Contiguous load Q0, address = R0 + 0
VLDRW.32            Q1, [R0, #(1*16)]         // Contiguous load Q1, address = R0 + 16
VLDRW.32            Q2, [R0, #(2*16)]         // Contiguous load Q2, address = R0 + 32
VLDRW.32            Q3, [R0, #(3*16)]         // Contiguous load Q3, address = R0 + 48
                                              // 4x4 float32_t matrix transpose alt.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VST40.32            {Q0, Q1, Q2, Q3}, [R1]    // Store interleaved-4 part 1
VST41.32            {Q0, Q1, Q2, Q3}, [R1]    // Store interleaved-4 part 2
VST42.32            {Q0, Q1, Q2, Q3}, [R1]    // Store interleaved-4 part 3
VST43.32            {Q0, Q1, Q2, Q3}, [R1]    // Store interleaved-4 part 4
                                              // R1 memory contains {
                                              // 0.0, 0.4, 0.8, 1.2,...
                                              // 0.1, 0.5, 0.9, 1.3,...
                                              // 0.2, 0.6, 1.0, 1.4,...
                                              // 0.3, 0.7, 1.1, 1.5 }
```

3.24.9 VSTR (System Register)

Store System Register.

Syntax

```
VSTR<c> <reg>, [Rn], #+/-<imm>
VSTR<c> <reg>, [Rn{, #+/-<imm>}]
VSTR<c> <reg>, [Rn, #+/-<imm>]!
```

Parameters

`Rn` The base register for the target address.

`c` See Standard Assembler Syntax Fields

`imm` The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4.

`reg` The system register to access This parameter must be one of the following values:

- `FPSCR`.
- `FPSCR_NZCVQC`.
- `VPR`.
- `P0`.
- `FPCXT_NS`.
- `FPCXT_S`.

Restrictions

- `reg` must not use `R0`
- `reg` must not use the same register as `R4` to `R7`
- `reg` must not use `R3`
- `Rn` must not use `PC`
- `reg` must not use the same register as `R8` to `R11`
- `Rn` must not use `SP` when the writeback variant is used

Post-conditions

There are no condition flags.

Operation

- Store a system register in memory. The target address is calculated from a base register plus an immediate offset.
- Access to the FPCXT payloads generates an UNDEFINED exception if the instruction is executed from Non-secure state.
- If CP10 is not enabled and either the Main extension is not implemented or the Floating-point context is active, access to FPCXT_NS will generate a NOCP UsageFault.
- Accesses to FPCXT_NS will not trigger lazy state preservation if there is no active Floating-point context.
- Accesses to FPCXT_NS do not trigger Floating-point context creation regardless of the value of FPCCR.ASprocessorN.
- FPSCR_nzcvqc allows access to FPSCR condition and saturation flags.
- The VPR register can only be accessed from privileged mode, otherwise the instruction behaves as a NOP.
- FPCXT_NS, enables saving and restoration of the Non-secure floating-point context.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- If the Floating-point context is active then the current FPSCR value is accessed and the default value in FPDSCR_NS is written into FPSCR, otherwise the default value in FPDSCR_NS is accessed.
- If neither the Floating-point extension nor MVE are implemented then access to this payload behaves as a NOP.
- FPCXT_S, enables saving and restoration of the Secure floating-point context.

VSTR (System Register) example

```
VSTR          P0,[SP,#0]          // Store VPR.p0 on the stack
VSTR          VPR,[SP]!           // Store VPR on the stack, post incremented load
VSTR          FPSCR,[SP], #4      // Store FPSCR on the stack
VSTR          FPSCR_NZCVQC,[SP,#8] // Store N, Z, C, V, and QC flags from FPSCR on the stack
```

3.24.10 VSTRB, VSTRH, VSTRW

Vector Store Register.

Syntax

```
VSTR{B,H,W}<v>.<dt> Qd, [Rn, #+/-<imm>]!
VSTR{B,H,W}<v>.<dt> Qd, [Rn], #+/-<imm>
VSTR{B,H,W}<v>.<dt> Qd, [Rn{, #+/-<imm>}]
```

Parameters

Qd Source vector register.

Rn The base register for the target address.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:
- 16
- 32

imm The signed immediate value that is added to base register to calculate the target address. No restrictions apply to the VSTRB variants. This value must be a multiple of 2 for the VSTRH variants. This value must be a multiple of 4 for the VSTRW variants.

v See Standard Assembler Syntax Fields

Restrictions

- rn must not use pc
- rn must not use sp when the writeback variant is used

Post-conditions

There are no condition flags.

Operation

Store consecutive elements to memory from a vector register. In indexed mode, the target address is calculated from a base register offset by an immediate value. Otherwise, the base register address is used directly. The sum of the base register and the immediate value can optionally be written back to the base register.

VSTRB example

```
MOV           R0, #0
VIDUP.U8      Q0, R0, #1          // Generates incrementing 8-bit sequence,
                                  // starting at 0 with increments of 1
VSTRB.S8      Q0, [R1, #8]        // Contiguous write, pre-index=8

MOV           R0, #0
VIDUP.U16     Q0, R0, #1          // Generates incrementing 16-bit sequence,
                                  // starting at 0 with increments of 1
VSTRB.S16     Q0, [R1, #0]        // Contiguous store + narrowing,
                                  // store 16-bit vector into 8-bit destination buffer

MOV           R0, #0
VIDUP.U32     Q0, R0, #1
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VSTRB.S32     Q0, [R1, #0]       // Contiguous store + narrowing,
                                 // store 32-bit vector into 8-bit destination buffer

VSTRB.S8      Q0, [R1, #8]!      // Contiguous store in R1 + 8 with write-back
                                 // Update R1 = R1 + 8 and store

VSTRB.S8      Q0, [R1], #8       // Contiguous store in R1, post-increment by 8
                                 // R1 = R1 + 8
```

VSTRH example

```
VSTRH.S16     Q0, [R1, #8]       // Contiguous store in R1 + 8  (= 4 half words offset)

VSTRH.S32     Q0, [R1]           // Contiguous store + narrowing,
                                 // store 32-bit vector into 16-bit destination buffer

VSTRH.S16     Q0, [R1, #8]!      // Contiguous store in R1 + 8 with write-back
                                 // R1 = R1 + 8

VSTRH.S16     Q0, [R1], #16      // Contiguous store, post-increment by 16
```

VSTRW example

```
VSTRW.S32     Q0, [R2, #8]       // Contiguous store in r2 + 8 (= 2 words offset)

VSTRW.S32     Q0, [R2, #16]!     // Contiguous store in R2 + 16 with write-back
                                 // R2 = R2 + 16

VSTRW.S32     Q0, [R2], #16      // Contiguous store in R2, post-increment by 16
                                 // R2 = R2 + 16

VSTRW.32      Q0, [SP, #16]
```

3.24.11 VSTRB, VSTRH, VSTRW, VSTRD (vector)

Vector Scatter Store.

Syntax

```
VSTRB<v>.<dt> Qd, [Rn, Qm]
VSTR{H,W,D}<v>.<dt> Qd, [Rn, Qm{, UXTW #os}]
VSTR{W,D}<v>.<dt> Qd, [Qm{, #+/-<imm>}]
VSTR{W,D}<v>.<dt> Qd, [Qm{, #+/-<imm>}]!
```

Parameters

Qd Source vector register.

Qm The base register for the target address. Vector offset register. The elements of this register contain the unsigned offsets to add to the base address.

Rn The base register for the target address.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:
- 8
- 16
- 32
- 64

imm The signed immediate value that is added to base register to calculate the target address. This value must be a multiple of 4, for the VSTRW variant. This value must be a multiple of 8, for the VSTRD variant.

os The amount by which the vector offset is left shifted by before being added to the general-purpose base address. If the value is present it must correspond to memory transfer size (1=half word, 2=word, 3=double word). This parameter must be one of the following values:
- 0
- 1

v See Standard Assembler Syntax Fields

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

- `Rn` must not use `PC`
- `Rn` must not use `SP` when the writeback variant is used

Post-conditions

There are no condition flags.

Operation

Store data from elements of Q[d] into a memory byte, halfword, word, or doubleword at the address contained in either of the following:

- A base register R[n] plus an offset contained in each element of Q[m], optionally shifted by the element size.
- Each element of Q[m] plus an immediate offset. The base element can optionally be written back, irrespective of predication, with that value incremented by the immediate or by the immediate scaled by the memory element size.

VSTRB (vector) example

```
MOV          R2, #1
VIDUP.u8     Q1, R2,#1          // Generates 8-bit incrementing sequence,
                                // starting at 1 with increments of 1
VSTRB.s8     Q0, [R0, Q1]       // Byte scatter store, base in R0, indexes in Q1

MOV          R2, #1
VIDUP.U16    Q1, R2,#1          // Generates 16-bit incrementing sequence,
                                // starting at 1 with increments of 1
VSTRB.S16    Q0, [R0, Q1]       // Halfword scatter store with narrowing, base in R0,
indexes in Q1
```

VSTRH (vector) example

```
MOV          R2, #0
VIDUP.U16    Q1,R2,#1              // Generates  16-bit incrementing sequence,
                                   // starting at 0 with increments of 1
VSTRH.S16    Q0, [R0, Q1, UXTW #1] // Halfword scatter store, base in R0,
                                    // indexes in q1, scaling applied (x2)

MOV          R2, #0
VIDUP.U32    Q1,R2,#1              // Generates 32-bit incrementing sequence,
                                   // starting at 0 with increments of 1
VSTRH.S32    Q0, [R0, Q1, UXTW #1] // Halfword scatter store with narrowing, base in R0,
                                   // indexes in Q1, scaling applied (x2)
```

VSTRW (vector) example

```
MOV          R2, #0
VIDUP.U32    Q1,R2,#1               // Generates 32-bit incrementing sequence,
                                    // starting at 0 with increments of 1
VSTRW.S32    Q0, [R0, Q1, UXTW #2]  // Word scatter store, base in R0, indexes in Q1,
                                    // scaling applied (x4)
MOV          R2, #0
VIDUP.U32    Q1,R2, #8          // Generates incrementing sequence,
                                // starting at 0 with increments of 8
VADD.S32     Q1, Q1, R0         // R0 contains base address,
                                // Q1 = vector of addresses = [R0 + 0, R0 + 8, R0 + 16, R0
+ 24]
VSTRW.S32    Q0, [Q1, #0]       // Word scatter store, base in Q1, pre-index=0

MOV          R2, #0
VIDUP.U32    Q1,R2, #8          // Generates incrementing sequence,
                                // starting at 0 with increments of 8
VADD.S32     Q1, Q1, R0         // R0 contains base address,
                                // Q1 = vector of addresses = [R0 + 0, R0 + 8, R0 + 16, R0
+ 24]
VSTRW.S32    Q0, [Q1, #16]!     // Word scatter store, bases in Q1, pre-index=16,
                                // write-back applied
                                // Store in [R0 + 16, R0 + 24, R0 + 32, R0 + 40]
```

VSTRD (vector) example

```
MOVS         R0, #1                // Build Q1 = {1LL, 2LL}
MOVS         R2, #2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMOV.s32      Q1, #0
VMOV          Q1[2], Q1[0], R0, R2
VSTRD.s64     Q0, [R0, Q1, UXTW #3] // Double word scatter store, base in R0,
                                    // indexes in Q1, scaling applied (x8)
VSTRD.s64     Q0, [Q1, #16]         // Double word scatter store, base in Q1, pre-index=16
VSTRD.s64     Q0, [Q1, #64]!        // Double word scatter store, base in Q1,
                                    // pre-index=64, write-back applied
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.25 Arm®v8.1-M vector move operation instructions

Reference material for the Cortex-M55 processor Armv8.1-M vector move operation instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.25.1 List of Arm®v8.1-M vector move operation instructions

An alphabetically ordered list of the Armv8.1-M vector move operation instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-19 Armv8.1-M vector move operation instructions

Mnemonic	Brief description	See
VMOV	VMOV (two 32 bit vector lanes to two general-purpose registers) Vector Move (two 32 bit vector lanes to two general-purpose registers).	*3.25.2 VMOV (two 32 bit vector lanes to two general-purpose registers)* on page 3-339
VMOV	VMOV (two general-purpose registers to two 32 bit vector lanes) Vector Move (two general-purpose registers to two 32 bit vector lanes).	*3.25.3 VMOV (two general-purpose registers to two 32 bit vector lanes)* on page 3-340

3.25.2 VMOV (two 32 bit vector lanes to two general-purpose registers)

Vector Move (two 32 bit vector lanes to two general-purpose registers).

Syntax

```
VMOV<c> Rt, Rt2, Qd[idx], Qd[idx2]
```

Parameters

Qd Source vector register.

Rt Destination general-purpose register

Rt2 Destination general-purpose register

c See Standard Assembler Syntax Fields

idx The first index for the vector register. This parameter must be one of the following values:
- 2.
- 3.

idx2 The second index for the vector register. This must be two less than the first index. This parameter must be one of the following values:
- 0.
- 1.

Restrictions

- Rt must not use the same register as Rt2
- Rt2 must not use the same register as SP and PC
- Rt must not use the same register as SP and PC

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

Copy two 32 bit vector lanes to two general-purpose registers.

VMOV (two 32 bit vector lanes to two general-purpose registers) example

```
// Move two 32-bit vector lanes to two general-purpose registers.
MOV            R2, #0
VIDUP.U32      Q0, R2, #1              // Generates incrementing sequence,
                                       // starting at 0 with step of 1
                                       // Q0 = [ 0 1 2 3 ]

VMOV           R0, R2, Q0[2], Q0[0]    // R0 = Q0[0], R2 = Q0[2]
                                       // R0 = 0, R2 = 2

VMOV           R1, R3, Q0[3], Q0[1]    // R1 = Q0[1], R3 = Q0[3]
                                       // R1 = 1, r3 = 3
```

3.25.3 VMOV (two general-purpose registers to two 32 bit vector lanes)

Vector Move (two general-purpose registers to two 32 bit vector lanes).

Syntax

```
VMOV<c> Qd[idx], Qd[idx2], Rt, Rt2
```

Parameters

Qd Destination vector register.

Rt Source general-purpose register.

Rt2 Source general-purpose register.

c See Standard Assembler Syntax Fields

idx The first index for the vector register. This parameter must be one of the following values:
- 2.
- 3.

idx2 The second index for the vector register. This must be two less than the first index. This parameter must be one of the following values:
- 0.
- 1.

Restrictions

- Rt2 must not use the same register as SP and PC
- Rt must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Copy two general-purpose registers to two 32 bit vector lanes.

VMOV (two general-purpose registers to two 32 bit vector lanes) example

```
// Move twp general-purpose registers to two 32-bit vector lanes
MOVS           R0, #0
MOVS           R1, #1
MOVS           R2, #2
MOVS           R3, #3

VMOV           Q0[2], Q0[0], R0, R2    // Q0[0] = R0, Q0[2] = R2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMOV            Q0[3], Q0[1], R1, R3    // Q0[1] = R1, Q0[3] = R3
                                        // Q0 = [ 0 1 2 3 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.26 Arm®v8.1-M RAS instruction

Reference material for the Cortex-M55 processor Armv8.1-M *Reliability, Availability, and Serviceability* (RAS) instruction set.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.26.1 ESB

Error Synchronization Barrier.

Syntax

```
ESB<c>
```

Parameters

c See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

This instruction is used to synchronize any asynchronous RAS exceptions. That is, RAS errors notified to the processor will not silently propagate past this instruction.

ESB example

```
Vector:                         // Exception handler entry
ESB                             // Error Synchronization Barrier
SUB         SP, SP, #14*4
PUSH        {R0-R12, LR}        // Save General-purpose registers
MOVW        R0, #0xEF04
MOVT        R0, #0xE002         // R0 = 0xE000EF04

LDR         R1, [R0]            // Read RAS Fault Status Register
PUSH        {R1}                // Save syndrome information
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.27 Arm®v8.1-M vector floating-point conversion and rounding operation instructions

Reference material for the Cortex-M55 processor Armv8.1-M vector floating-point conversion and rounding operation instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.27.1 List of Arm®v8.1-M vector floating-point conversion and rounding operation instructions

An alphabetically ordered list of the Armv8.1-M vector floating-point conversion and rounding operation instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-20 Armv8.1-M vector floating-point conversion and rounding operation instructions

Mnemonic	Brief description	See
VCVT	VCVT (between floating-point and fixed-point) Vector Convert between floating-point and fixed-point	*3.27.2 VCVT (between floating-point and fixed-point)* on page 3-343
VCVT	VCVT (between floating-point and integer) Vector Convert between floating-point and integer	*3.27.3 VCVT (between floating-point and integer)* on page 3-344
VCVT	VCVT (between single and half-precision floating-point) Vector Convert between half-precision and single-precision	*3.27.4 VCVT (between single and half-precision floating-point)* on page 3-345
VCVT	VCVT (from floating-point to integer) Vector Convert from floating-point to integer	*3.27.5 VCVT (from floating-point to integer)* on page 3-346

3.27.2 VCVT (between floating-point and fixed-point)

Vector Convert between floating-point and fixed-point.

Syntax

```
VCVT<v>.<dt> Qd, Qm, #<fbits>
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt This parameter must be one of the following values:
- Convert signed 16-bit integer to half-precision floating-point, `F16.S16`.
- Convert unsigned 16-bit integer to half-precision floating-point, `F16.U16`.
- Convert half-precision floating-point to signed 16-bit integer, `S16.F16`.
- Convert half-precision floating-point to unsigned 16-bit integer, `U16.F16`.
- Convert signed 32-bit integer to single-precision floating-point, `F32.S32`.
- Convert unsigned 32 bit integer to single-precision floating-point, `F32.U32`.
- Convert single-precision floating-point to signed 32 bit integer, `S32.F32`.
- Convert single-precision floating-point to unsigned 32 bit integer, `U32.F32`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

fbits The number of fraction bits in the fixed-point number. For 16-bit fixed-point, this number must be in the range 1-16. For 32-bit fixed-point, this number must be in the range 1-32. The value of (64 - `fbits`) is encoded in `imm6`.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Convert between floating-point and fixed-point values in elements of a vector register. The number of fractional bits in the fixed-point value is specified by an immediate. Fixed-point values can be specified as signed or unsigned. The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode. For floating-point to fixed-point operation, if the source value is outside the range of the target fixed-point type, the result is saturated.

VCVT (between floating-point and fixed-point) example

```
 // Convert Q31 vector to single precision float vector (incrementing + sign-alternating
sequence)
MOV           R0, #2           // Wrap sequence start
MOV           R1, #4           // Wrap sequence top
MOV           R2, #1
VIWDUP.U32    Q0, r0, R1, #2   // Alternating {0, 2} sequence generation
VSUB.S32      Q0, Q0, R2       // Alternating {+1, -1} sequence generation
MOV           R0, #0x10000000
VMUL.S32      Q0, Q0, R0       // Alternating {+0x10000000, -0x10000000} sequence
MOV           R0, #1
VIDUP.U32     Q1, R0, #2       // Incrementing sequence, starting at 1, increment of 2
VMUL.S32      Q0, Q0, Q1
                              // Q0 = 2 ^ 31  * [ -0.1250 0.3750 -0.6250 0.8750 ]
                              //     = [ -268435456 805306368 -1342177280 1879048192 ]
VCVT.F32.S32  Q1,Q0,#31       // Q1[i] = (float31_t)Q[0] / (float31_t)2^31  i={0..3}
                              // Q1 = [ -0.125 0.375 -0.625 0.875 ]

// Convert Q15 vector to half precision float vector (incrementing + sign-alternating
sequence)
MOV           R0, #2           // Wrap sequence start
MOV           R1, #4           // Wrap sequence top
MOV           R2, #1
VIWDUP.U16    Q0, R0, R1, #2   // Alternating {0, 2} sequence generation
VSUB.S16      Q0, Q0, R2       // Alternating {+1, -1} sequence generation
MOV           R0, #0x1000
VMUL.S16      Q0, q0, r0       // Alternating {+0x1000, -0x1000} sequence
MOV           R0, #1
VIDUP.U16     Q1, R0, #2       // Incrementing sequence, starting at 1, increment of 2
VMUL.S16      Q0, Q0, Q1
                               // Q0 = 2 ^ 15 * [  -0.1250 0.3750  -0.6250 0.8750  0.8750
-0.6250  0.3750 -0.1250 ] %
                               //    = [  -4096 12288  -20480 28672  28672 -20480  12288
-4096  ]
VCVT.F16.S16  Q1, Q0, #15      // Q1[i] = (float16_t)Q[0] / (float16_t)2^15  i={0..7}
                               // Q1 = [ -0.125 0.375  -0.625 0.875  0.875 -0.625  0.375
-0.125
```

3.27.3 VCVT (between floating-point and integer)

Vector Convert between floating-point and integer.

Syntax

```
VCVT<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Qm Source vector register.

dt This parameter must be one of the following values:

- `F16.S16`.
- Convert signed 16 bit integer to half-precision floating-point `F32.S32`.
- Convert signed 32 bit integer to single-precision floating-point `F16.U16`.
- Convert unsigned 16 bit integer to half-precision floating-point `F32.U32`.
- Convert unsigned 32 bit integer to single-precision floating-point `S16.F16`.
- Convert half-precision floating-point to signed 16 bit integer `S32.F32`.
- Convert single-precision floating-point to signed 32 bit integer `U16.F16`.
- Convert half-precision floating-point to unsigned 16 bit integer `U32.F32`.
- Convert single-precision floating-point to unsigned 32 bit integer.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Convert between floating-point and integer values in elements of a vector register. When converting to integer the value is rounded towards zero, when converting to floating-point the value is rounded to nearest. For floating-point to integer operation, if the source value is outside the range of the target integer type, the result is saturated.

VCVT (between floating-point and integer) example

```
// Vector Convert between 32-bit floating-point and integer back and forth
MOV           R0, #0
VIDUP.u32     Q0, R0, #2        // Incrementing sequence, starting at 0, increment of 2
                                // Q0 = [ 0 2 4 6 ]
VCVT.F32.S32  Q1, Q0            // Q1[i] = (float32_t)q0[i]  i={0..3} 32-bit integer to F32
vector conversion
                                // Q1 = [ 0.000 2.000 4.000 6.000 ]

VCVT.S32.F32  Q2, Q1            // Q2[i] = (int32_t)Q1[i]    i={0..3} F32 to 32-bit integer
conversion
                                // Q2 = [ 0 2 4 6 ]
```

3.27.4 VCVT (between single and half-precision floating-point)

Vector Convert between half-precision and single-precision.

Syntax

```
VCVT<T><v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

T Specifies that the FP16 value read from or written to the top or bottom half of the FP32 vector register element. This parameter must be one of the following values:

- `B`, indicates bottom half.
- `T`, indicates top half.

dt This parameter must be one of the following values:

- Convert single-precision to half-precision, `F16.F32`.
- Convert half-precision to single-precision, `F32.F16`.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Convert between half-precision and single-precision floating-point values in elements of a vector register. For half-precision to single-precision operation, the top half (T variant) or bottom half (B variant) of the source vector register is selected. For single-precision to half-precision operation, the top half (T variant) or bottom half (B variant) of the destination vector register is selected and the other half retains its previous value.

VCVT (between single and half-precision floating-point) example

```
// Convert two single-precision vectors to half-precision vector
VLD20.32       {Q0, Q1}, [R0]  // Vector de-interleaving load
VLD21.32       {Q0, Q1}, [R0]  // In this case, R0 points to float32_t array containing
                               // { .0f, .1f, .2f, .3f...}
                               // Q0 = [ 0.000 0.200 0.400 0.600 ]        % VLD2 even
                               // Q1 = [ 0.100 0.300 0.500 0.700 ]        % VLD2 odd
VCVTB.F16.F32  Q2, Q0          // Convert Q0 F32 elements into F16 and place into bottom
parts of Q2
                               // Q2[2*i] = (float16_t)Q0[i]  i={0..3}
                               // Q2 = [ 0.000 x  0.200 x  0.400 x  0.600 x ]
                               // (x indicates the unchanged parts of the vector)
VCVTT.F16.F32  Q2, Q1          // Convert F32 Q1 into top part of Q2
                               // Q2[2*i+1] = (float16_t)Q1[i]  i={0..3}
                               // Q2 = [ x 0.100  x 0.300 x 0.500  x 0.700 ]
                               // (x indicates the unchanged parts of the vector)

// Convert half-precision vector bottom elements into single precision vector
VLDRH.U32      Q0, [R0]        // Load 16-bit memory into 32-bit vector (widening)
                               // R0 points to float16_t array containing
                               // .0, .1, .2, .3, .4, .5, .6, .7, ..
                               // Q0 = [ 0.000 0.000  0.100 0.000  0.200 0.000  0.300
0.000 ]
VCVTB.F32.F16  Q1, Q0          // Convert bottom parts of Q0 in F32 vectors
                               // Q1[i] = (float32_t)Q0[2*i]  i={0..3}
                               // Q1 = [ 0.000 0.100 0.200 0.300 ]
```

3.27.5 VCVT (from floating-point to integer)

Vector Convert from floating-point to integer.

Syntax

```
VCVT<ANPM><v>.<dt> Qd, Qm
```

Parameters

ANPM The rounding mode. This parameter must be one of the following values:
- A, round to nearest with ties to away.
- N, round to nearest with ties to even.
- P, round towards plus infinity.
- M, round towards minus infinity.

Qd Destination vector register.

Qm Source vector register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter must be one of the following values:

- Convert half-precision floating-point to signed 16 bit integer, `S16.F16`.
- Convert single-precision floating-point to signed 32 bit integer, `S32.F32`.
- Convert half-precision floating-point to unsigned 16 bit integer, `U16.F16`.
- Convert single-precision floating-point to unsigned 32 bit integer, `U32.F32`.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Convert each element in a vector from floating-point to integer using the specified rounding mode and place the results in a second vector. If a source element is outside the range of the target integer type, the result element is saturated.

VCVT (from floating-point to integer) example

```
// Convert half-precision floating point vector to Q15 vector
MOV           R0, #1             // Wrap sequence bottom
MOV           R1, #4             // Wrap sequence top
MOV           R3, #0x150F       // 0.00123456f16
VIDUP.U16     Q0, R0, #1         // Incrementing sequence, starting at 1, increment of 1
VCVT.F16.S16  Q0, Q0             // Convert 16-bit integer to half-precision floating-point
VMUL.F16      Q0, Q0, R3         // Q0[i] = Q0[i] * 0.00123456

MOV           R3, #0x7800       // 32767.0f16 (Q.15 conversion)
VMUL.F16      Q1,Q0,R3           // Q1[i] = Q0[i] * 32767.0f16
                                 // Q0[i] = [ 0.001 0.002  0.004 0.005  0.006 0.007  0.009
                                 // 0.010 ]
                                 // Q1[i] = [ 40.469 80.938  121.375 161.875  202.375
242.750
                                 // 283.250 323.750 ]
VCVTA.S16.F16 Q2,Q1              // Q1[i] = (int16_t)RND(q0[i]) Round to
                                 // Nearest with Ties to Away rounding i={0..7}
                                 // Q3[i] = [  40 81  121 162  202 243  283 324  ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.28 Arm®v8.1-M security instructions

Reference material for the Cortex-M55 processor Armv8.1-M security instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.28.1 List of Arm®v8.1-M security instructions

An alphabetically ordered list of the Armv8.1-M security instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-21 Armv8.1-M security instructions

Mnemonic	Brief description	See
CLRM	Clear multiple	*3.28.2 CLRM* on page 3-348
VSCCLRM	Floating-point Secure Context Clear Multiple	*3.28.3 VSCCLRM* on page 3-349

3.28.2 CLRM

Clear multiple.

Syntax

```
CLRM<c> <registers>
```

Parameters

c See Standard Assembler Syntax Fields.

registers A list of the registers to clear. The valid registers are APSR, LR/R14, and R0-R12.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Zeros the specified general-purpose registers. It is IMPLEMENTATION DEFINED whether this instruction is exception-continuable. If an exception returns to this instruction with non-zero EPSR.ICI bits, and the processor does not support exception-continuable behavior, the instruction restarts from the beginning. For more information on EPSR, see *Execution Program Status Register* on page 2-42.

CLRM example

```
// Clear multiple
// Flash accumulator clearing in 32-bit integer matrix x vector multiplication illustration

// Inner loop extract
// R6 points to vector base
// R7 points to matrix row N
// R8 points to matrix row N + 1
// R8 points to matrix row N + 2
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
// Flash clear six registers (3 x 64-bit accumulators in this examples)
CLRM            {R0, R1, R2, R3, R4, R5}
                                    // CLRM  {R0-R5} notation can also be used
                                    // R0 = R1 = R2 = R3 = R4 = R5 = 0
MOV             LR, #32             // Vector number of rows
WLSTP.32        LR, LR, LF          // While loop start with Tail Predication
2:
VLDRW.U32       Q0, [R6], #16       // Vector contiguous load + post-increment (vector)
VLDRW.U32       Q1, [R7], #16       // Vector contiguous load + post-increment (matrix row
N)
VMLALVA.S32     R0, R1, Q0, Q1      // R0:R1 += sum(Q0[i] * Q1[i]) i={0..3}
VLDRW.U32       Q2, [R8], #16       // Vector contiguous load + post-increment (matrix row N
+1)
VMLALVA.S32     R2, R3, Q0, Q2      // R2:R3 += sum(Q0[i] * Q1[i]) i={0..3}
VLDRW.U32       Q3, [R9], #16       // Vector contiguous load + post-increment (matrix row N
+2)
VMLALVA.S32     R4, R5, Q0, Q3      // R4:R5 += sum(q0[i] * Q1[i]) i={0..3}
LETP            LR, #2B             // Loop + tail predication end
1:
```

3.28.3 VSCCLRM

Floating-point Secure Context Clear Multiple.

Syntax

```
VSCCLRM<c> <sreglist>
VSCCLRM<c> <dreglist>
```

Parameters

c See Standard Assembler Syntax Fields

dreglist Is the list of consecutively numbered 64-bit floating-point registers to be cleared. Because this instruction always clears the VPR register, it is mandatory to have VPR in the register list

sreglist Is the list of consecutively numbered 32-bit floating-point registers to be cleared.This instruction always clears the VPR register, therefore, it is mandatory to have VPR in the register list

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Zeros VPR and the specified floating-point registers if there is an active floating-point context. This instruction is UNDEFINED if executed in Non-secure state. This instruction is present on all processors that implement the Armv8.1-M architecture, even if the Floating-point Extension is not present. It is IMPLEMENTATION DEFINED whether this instruction is exception-continuable. See EPSR.ICI. If an exception returns to this instruction with non-zero EPSR.ICI bits, and the processor does not support exception-continuable behavior, the instruction restarts from the beginning. If the Floating-point Extension is not implemented, access to the FPCXT payload is RES0.

VSCCLRM example

```
VSCCLRM         {S0, S1, S2, S3, VPR}  // Clear S0-S3, VPR
VSCCLRM         {S0-S7, VPR}           // Alternative notation clear S0-S7, VPR

VSCCLRM         {D0, D1, D2, D3, VPR}  // Clear D0-D3, VPR
VSCCLRM         {D0-D4, VPR}           // Alternative notation clear D0-S7, VPR

IT              HI
VSCCLRMHI       {S3-S31, VPR}          // Conditional clear of S3-S31, VPR
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.29 Arm®v8.1-M vector arithmetic instructions

Reference material for the Cortex-M55 processor Armv8.1-M vector arithmetic instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.29.1 List of Arm®v8.1-M vector arithmetic instructions

An alphabetically ordered list of the Armv8.1-M vector arithmetic instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-22 Armv8.1-M arithmetic instructions

Mnemonic	Brief description	See
VABAV	Vector Absolute Difference and Accumulate Across Vector	*3.29.2 VABAV* on page 3-352
VABD	Vector Absolute Difference	*3.29.3 VABD* on page 3-353
VABD	VABD (floating-point) Vector Absolute Difference	*3.29.4 VABD (floating-point)* on page 3-354
VABS	Vector Absolute	*3.29.5 VABS* on page 3-355
VABS	VABS (floating-point) Vector Absolute	*3.29.6 VABS (floating-point)* on page 3-355
VADC	Whole Vector Add With Carry	*3.29.7 VADC* on page 3-356
VADD	Vector Add	*3.29.8 VADD* on page 3-357
VADD	VADD (floating-point) Vector Add	*3.29.9 VADD (floating-point)* on page 3-358
VADDLV	Vector Add Long Across Vector	*3.29.10 VADDLV* on page 3-359
VADDV	Vector Add Across Vector	*3.29.11 VADDV* on page 3-360
VCADD	Vector Complex Add with Rotate.	*3.29.12 VCADD* on page 3-361
VCADD	VCADD (floating-point) Vector Complex Add with Rotate	*3.29.13 VCADD (floating-point)* on page 3-362
VCLS	Vector Count Leading Sign-bits	*3.29.14 VCLS* on page 3-363
VCLZ	Vector Count Leading Zeros	*3.29.15 VCLZ* on page 3-364
VCMLA	VCMLA (floating-point) Vector Complex Multiply Accumulate.	*3.29.16 VCMLA (floating-point)* on page 3-365
VCMUL	VCMUL (floating-point) Vector Complex Multiply	*3.29.17 VCMUL (floating-point)* on page 3-366
VDDUP, VDWDUP	Vector Decrement and Duplicate, Vector Decrement with Wrap and Duplicate	*3.29.18 VDDUP, VDWDUP* on page 3-368
VDUP	Vector Duplicate	*3.29.19 VDUP* on page 3-369
VFMA	VFMA (vector by scalar plus vector, floating-point) Vector Fused Multiply Accumulate	*3.29.20 VFMA (vector by scalar plus vector, floating-point)* on page 3-370

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-22 Armv8.1-M arithmetic instructions (continued)

Mnemonic	Brief description	See
VFMA, VFMS	VFMA, VFMS (floating-point) Vector Fused Multiply Accumulate, Vector Fused Multiply Subtract	*3.29.21 VFMA, VFMS (floating-point)* on page 3-371
VFMAS	VFMAS (vector by vector plus scalar, floating-point) Vector Fused Multiply Accumulate Scalar	*3.29.22 VFMAS (vector by vector plus scalar, floating-point)* on page 3-372
VHADD	Vector Halving Add	*3.29.23 VHADD* on page 3-372
VHCADD	Vector Halving Complex Add with Rotate	*3.29.24 VHCADD* on page 3-373
VHSUB	Vector Halving Subtract	*3.29.25 VHSUB* on page 3-375
VIDUP, VIWDUP	Vector Increment and Duplicate, Vector Increment with Wrap and Duplicate	*3.29.26 VIDUP, VIWDUP* on page 3-376
VMLA	VMLA Vector Multiply Accumulate	*3.29.27 VMLA (vector by scalar plus vector)* on page 3-376
VMLADAV	Vector Multiply Add Dual Accumulate Across Vector	*3.29.28 VMLADAV* on page 3-377
VMLALDAV	Vector Multiply Add Long Dual Accumulate Across Vector	*3.29.29 VMLALDAV* on page 3-378
VMLALV	Vector Multiply Accumulate Long Across Vector	*3.29.30 VMLALV* on page 3-380
VMLAS	VMLAS (vector by vector plus scalar)	*3.29.31 VMLAS (vector by vector plus scalar)* on page 3-380
VMLAV	Vector Multiply Accumulate Across Vector	*3.29.32 VMLAV* on page 3-381
VMLSDAV	Vector Multiply Subtract Dual Accumulate Across Vector	*3.29.33 VMLSDAV* on page 3-382
VMLSLDAV	Vector Multiply Subtract Long Dual Accumulate Across Vector	*3.29.34 VMLSLDAV* on page 3-383
VMUL	Vector Multiply	*3.29.35 VMUL* on page 3-384
VMUL	VMUL (floating-point) Vector Multiply	*3.29.36 VMUL (floating-point)* on page 3-385
VMULH, VRMULH	Vector Multiply Returning High Half, Vector Rounding Multiply Returning High Half	*3.29.37 VMULH, VRMULH* on page 3-386
VMULL	VMULL (integer) Vector Multiply Long	*3.29.38 VMULL (integer)* on page 3-387
VMULL	VMULL (polynomial) Vector Multiply Long	*3.29.39 VMULL (polynomial)* on page 3-388
VNEG	Vector Negate	*3.29.40 VNEG* on page 3-389
VNEG	VNEG (floating-point) Vector Negate	*3.29.41 VNEG (floating-point)* on page 3-390
VQABS	Vector Saturating Absolute	*3.29.42 VQABS* on page 3-390
VQADD	Vector Saturating Add	*3.29.43 VQADD* on page 3-391
VQDMLADH, VQRDMLADH	Vector Saturating Doubling Multiply Add Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Add Dual Returning High Half	*3.29.44 VQDMLADH, VQRDMLADH* on page 3-392

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-22 Armv8.1-M arithmetic instructions (continued)

Mnemonic	Brief description	See
VQDMLAH, VQRDMLAH	VQDMLAH, VQRDMLAH (vector by scalar plus vector) Vector Saturating Doubling Multiply Accumulate, Vector Saturating Rounding Doubling Multiply Accumulate	*3.29.45 VQDMLAH, VQRDMLAH (vector by scalar plus vector)* on page 3-393
VQDMLASH, VQRDMLASH	VQDMLASH, VQRDMLASH (vector by vector plus scalar) Vector Saturating Doubling Multiply Accumulate Scalar High Half, Vector Saturating Rounding Doubling Multiply Accumulate Scalar High Half	*3.29.46 VQDMLASH, VQRDMLASH (vector by vector plus scalar)* on page 3-394
VQDMLSDH, VQRDMLSDH	Vector Saturating Doubling Multiply Subtract Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Subtract Dual Returning High Half.	*3.29.47 VQDMLSDH, VQRDMLSDH* on page 3-395
VQDMULH, VQRDMULH	Vector Saturating Doubling Multiply Returning High Half, Vector Saturating Rounding Doubling Multiply Returning High Half	*3.29.48 VQDMULH, VQRDMULH* on page 3-396
VQDMULL	Vector Multiply Long	*3.29.49 VQDMULL* on page 3-398
VQNEG	Vector Saturating Negate	*3.29.50 VQNEG* on page 3-399
VQSUB	Vector Saturating Subtract	*3.29.51 VQSUB* on page 3-400
VREV16	Vector Reverse	*3.30.18 VREV16* on page 3-421
VREV32	Vector Reverse	*3.30.19 VREV32* on page 3-422
VREV642	Vector Reverse	*3.30.20 VREV64* on page 3-423
VRHADD	Vector Rounding Halving Add	*3.29.52 VRHADD* on page 3-400
VRINT	VRINT (floating-point)	*3.29.53 VRINT (floating-point)* on page 3-401
VRMLALDAVH	Vector Rounding Multiply Add Long Dual Accumulate Across Vector Returning High 64 bits.	*3.29.54 VRMLALDAVH* on page 3-402
VRMLALVH	Vector Multiply Accumulate Long Across Vector Returning High 64 bits	*3.29.55 VRMLALVH* on page 3-404
VRMLSLDAVH	Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bits	*3.29.56 VRMLSLDAVH* on page 3-404
VSBC	Whole Vector Subtract With Carry	*3.29.57 VSBC* on page 3-405
VSUB	Vector Subtract	*3.29.58 VSUB* on page 3-406
VSUB	VSUB (floating-point) Vector Subtract	*3.29.59 VSUB (floating-point)* on page 3-407

3.29.2 VABAV

Vector Absolute Difference and Accumulate Across Vector.

Syntax

```
VABAV<v>.<dt> Rda, Qn, Qm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qm Second source vector register.

Qn First source vector register.

Rda General-purpose source and destination register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

`Rda` must not use the same register as `SP` and `PC`.

Post-conditions

There are no condition flags.

Operation

Subtract the elements of the second source vector register from the corresponding elements of the first source vector and accumulate the absolute values of the results. The initial value of the general-purpose destination register is added to the result.

VABAV example

```
// Vector Absolute Difference between 2 incrementing 16-bit integer sequences with
Accumulate Across Vector
MOVS          R0, #0      // Incrementing sequence start
MOVS          R2, #100    // R2 = 100 (accumulator)
VIDUP.U16     Q0, R0, #2  // 16-bit incrementing sequence starting at 0, increment of 2
VIDUP.U16     Q1, R0, #1  // 16-bit incrementing sequence starting at 16
                          // (continuation of previous VIDUP), increment of 1
                          // Q0 = [  0 2  4 6  8 10  12 14  ]
                          // Q1 = [  16 17  18 19  20 21  22 23  ]
VABAV.S16     R2, Q0, Q1  // R2 = 100 + sum(|Q0[i] - Q1[i]|),  i={0..7}
                          // R2 =  100 + |-16| + |-15| + |-14| + |-13| + |-12| + |-11| +
|-10| + |-9| = 200
```

3.29.3 VABD

Vector Absolute Difference.

Syntax

```
VABD<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:
- `S8`
- `U8`
- `S16`
- `U16`
- `S32`
- `U32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Subtract the elements of the second source vector register from the corresponding elements of the first source vector register and place the absolute values of the results in the elements of the destination vector register.

VABD example

```
// Vector Absolute Difference between 2 incrementing 32-bit integer sequences
MOVS          R0, #0      // Incrementing sequence start
MOVS          R2, #100    // R2 = 100 (second incrementing sequence start + accumulator)
VIDUP.U32     Q0,R0,#2    // 32-bit incrementing sequence starting at 0, increment of 2
VIDUP.U32     Q1,R2,#1    // 32-bit incrementing sequence starting at 100, increment of 1
                          // Q0 = [ 0 2 4 6 ]
                          // Q1 = [ 100 101 102 103 ]
VABD.S32      Q2, Q0, Q1  // Q2[i] = |Q0[i] - Q1[i]|  i={0..3}
                          // Q2 = [ |-100| |-99| |-98| |-97| ] = [ 100 99 98 97 ]
```

3.29.4 VABD (floating-point)

Vector Absolute Difference.

Syntax

```
VABD<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  — `F32`
  — `F16`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Subtract the elements of the second source vector from the corresponding elements of the first source vector and place the absolute values of the results in the elements of the destination vector register.

VABD (floating-point) example

```
// Vector Absolute Difference between 2 incrementing 32-bit float sequences
MOVS          R0, #0       // 1st incrementing sequence start
MOVS          R2, #100     // 2nd incrementing sequence start
VIDUP.U32     Q0, R0, #2   // Incrementing sequence starting at 0, increment of 2
VCVT.F32.S32  Q0, Q0       // 32-bit integer to simple precision float conversion
VIDUP.U32     Q1, R2, #1   // Incrementing sequence starting at 100, increment of 1
VCVT.F32.S32  Q1, Q1       // 32-bit integer to simple precision float conversion
                           // Q0 = [ 0.000 2.000 4.000 6.000 ]
                           // Q1 = [ 100.0 101.0 102.0 103.0 ]
VABD.F32      Q2, Q0, Q1   // Q2[i] = |Q0[i] - Q1[i]|  i={0..3}
                           // Q2 = [ |-100.0| |-99.0| |-98.0| |-97.0| ]
                           // = [ 100.0 99.0 98.0 97.0 ]
```

3.29.5 VABS

Vector Absolute.

Syntax

```
VABS<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - S8
  - S16
  - S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute the absolute value of each element in a vector register.

VABS example

```
// Compute the absolute value of each element in an alternating {+1/-1} 16-bit integer
vector register.
MOV           R2, #1
MOV           R0, #0             // Incrementing + wrapping sequence bottom
MOV           R1, #4             // incrementing + wrapping sequence top
VIWDUP.U16    Q0, R0, R1, #2     // Alternating {0, 2} sequence generation
VSUB.S16      Q0, Q0, R2         // Alternating {-1, +1} sequence = [0 2 0 2 0 2 0 2] - 1
                                 // Q0 = [ -1 1  -1 1  -1 1  -1 1  ]
VABS.S16      Q1, Q0             // Q1[i] = |Q0[i]|  i={0..7}
                                 // Q1 = [  1 1  1 1  1 1  1 1  ]
```

3.29.6 VABS (floating-point)

Vector Absolute.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VABS<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - F16
  - F32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute the absolute value of each element of a vector register.

VABS (floating-point) example

```
// Compute the absolute value of each element in an alternating {+1.0/-1.0} 32-bit float
vector register.
MOV           R2, #1
MOV           R0, #0            // Incrementing + wrapping sequence bottom
MOV           R1, #4            // Incrementing + wrapping sequence top
VIWDUP.U32    Q0, R0, R1, #2    // Alternating {0, 2} sequence
VSUB.S32      Q0, Q0, R2        // Alternating {-1, +1} sequence
VCVT.F32.S32  Q0, Q0            // 32-bit integer to simple precision float
                                // conversion
                                // Q0 = [-1.000 1.000 -1.000 1.000 ]
VABS.F32      Q1, Q0            // Q1[i] = |Q0[i]| i={0..3}
                                // Q1 = [ 1.000 1.000 1.000 1.000 ]
```

3.29.7 VADC

Whole Vector Add With Carry.

Syntax

```
VADC{I}<v>.I32 Qd, Qn, Qm
```

Parameters

I Specifies where the initial carry in for wide arithmetic comes from. This parameter must be one of the following values:
- An unlisted parameter, encoded as `I=0`, which indicates that the carry input comes from FPSCR.C.
- `I=1`, which indicates carry input is 0.

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

v See Standard Assembler Syntax Fields

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

There are no restrictions.

Post-conditions

Reads C flag in FPSCR and updates the N, Z, C, and V flags in FPSCR register.

Operation

Add with carry across beats, with carry in from and out to FPSCR.C. Initial value of FPSCR.C can be overridden by using the VADCI variant. FPSCR.C is not updated for beats disabled because of predication. FPSCR.N, FPSCR.V and FPSCR.Z are zeroed.

VADC example

```
// Whole Vector Add With Carry, input carry cleared
VLDRW.32       Q0, [R0]    // Contiguous 32-bit vector load
VLDRW.32       Q1, [R1]    // Contiguous 32-bit vector load
                           // Q0 = [ 0x10000000 0x20000000 0x30000000 0x40000000 ]
                           // = 0x40000000300000002000000010000000
                           // Q1 = [ 0x10000000 0xF0000000 0x50000000 0x70000000 ]
                           // = 0x7000000050000000F000000010000000

VADCI.I32      Q2, Q0, Q1  // Q2 = Q0 + Q1  (carry input = 0)
                           // Q2 = [ 0x20000000 0x10000000 0x80000001 0xB0000000 ]
                           // = 0xB0000000800000011000000020000000


// Whole Vector Add With Carry, input carry read from current FPSCR
VLDRW.32       Q0, [R0]     // Contiguous 32-bit vector load
VLDRW.32       Q1, [R1]     // Contiguous 32-bit vector load
                            // Q0 = [ 0x10000000 0x20000000 0x30000000 0x40000000 ]
                            // = 0x40000000300000002000000010000000
                            // Q1 = [ 0xEFFFFFFF 0xF0000000 0x50000000 0x70000000 ]
                            // = 0x7000000050000000F0000000EFFFFFFF

VADC.I32       Q2, Q0, Q1   // Q2 = Q0 + Q1 + FPSCR.C
                            // Assumes input carry is set
                            // Q2 = [ 0x00000000 0x10000001 0x80000001 0xB0000000 ]
                            // = 0xB0000000800000011000000100000000
```

3.29.8 VADD

Vector Add.

Syntax

```
VADD<v>.<dt> Qd, Qn, Qm
VADD<v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - I8
  - I16
  - I32

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use the same register as `SP` and `PC`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is then written to the destination vector register.

VADD example

```
// Addition of 2 x 8-bit integer vectors
MOV           R0, #1
MOV           R2, #1
VIDUP.U8      Q0, R0, #1   // Generates 8-bit incrementing sequence,
                           // starting at 1 with increments of 1
VIDUP.U8      Q1, Q2, #4   // Generates 8-bit incrementing sequence,
                           // starting at 1 with increments of 4
                           // Q0 = [  1 2 3 4  5 6 7 8  9 10 11 12  13 14 15 16 ]
                           // Q1 = [  1 5 9 13  17 21 25 29  33 37 41 45  49 53 57 61 ]
VADD.I8       Q2, Q0, Q1   // Q2[i] = Q0[i] + Q1[i]   i={0..15}
                           // Q2 = [  2 7 12 17  22 27 32 37  42 47 52 57  62 67 72 77 ]

// Addition of a 16-bit integer vector with a scalar
MOV           R2, #1
MOV           R0, #0            // Wrap sequence bottom
MOV           R1, #4            // Wrap sequence top
MOV           R3, #4000         // Offset
VIWDUP.U16    Q0, R0, R1, #2    // Alternating {0, 2} sequence generation
VSUB.I16      Q0, Q0, R2        // Alternating {-1, +1} sequence
                                // Q0 = [  -1 1  -1 1  -1 1  -1 1  ]
VADD.I16      Q1, Q0, R3        // Q1[i] = Q0[i] + R3   i={0..7}
                                // Q1 = [  3999 4001  3999 4001  3999 4001  3999 4001  ]

// Addition of 2 x 16-bit integer vectors
VIDUP.U16     Q1, R2, #4      // Generates 16-bit incrementing sequence,
                              // starting at 1 with increments of 4
                              // Q0 = [  -1 1  -1 1  -1 1  -1 1  ]
                              // Q1 = [  1 5  9 13  17 21  25 29  ]
VADD.I16      Q2, Q0, Q1      // Q2[i] = Q0[i] + Q1[i]   i={0..7}
                              // Q2 = [  0 6 8 14  16 22  24 30  ]
```

3.29.9 VADD (floating-point)

Vector Add.

Syntax

```
VADD<v>.<dt> Qd, Qn, Qm
VADD<v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use the same register as `SP` and `PC`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register.

VADD (floating-point) example

```
// Addition of a 16-bit float vector with a scalar
MOV           R2, #1
MOV           R0, #0            // Wrap sequence bottom
MOV           R1, #4            // Wrap sequence top
VIWDUP.U16    Q0, R0, R1, #2    // Alternating {0, 2} sequence
VSUB.S16      Q0, Q0, r2        // Alternating {-1, +1} sequence
VCVT.F16.S16  Q0, Q0            // Convert 16-bit integer to half precision float vector
MOVW          R0, #0x5100       // R0 = 40.0f16 offset
                                // Q0 = [ -1.000 1.000  -1.000 1.000
                                // -1.000 1.000  -1.000 1.000 ]
VADD.F16      Q2, Q0, R0        // Q1[i] = Q0[i] + R0   i={0..7}
                                // Q2 = [ 39.000 41.000  39.000 41.000
                                // 39.000 41.000  39.000 41.000  ]

// Addition of 2 x 16-bit float vectors
VIDUP.U16     Q1, R2, #4      // Generates incrementing sequence,
                              // starting at 1 with increments of 4
VCVT.F16.S16  Q1, Q1          // Convert 16-bit integer to half precision float vector
                              // Q0 = [ -1.000 1.000  -1.000 1.000
                              // -1.000 1.000  -1.000 1.000 ]
                              // Q1 = [ 1.000 5.000  9.000 13.000
                              // 17.000 21.000  25.000 29.000 ]
VADD.F16      Q2, Q0, Q1      // Q2[i] = q0[i] + q1[i]   i={0..7}
                              // Q2 = [ 0.000 6.000  8.000 14.000
                              // 16.000 22.000  24.000 30.000 ]
```

3.29.10 VADDLV

Vector Add Long Across Vector.

Syntax

```
VADDLV{A}<v>.<dt> RdaLo, RdaHi, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- An unlisted parameter, encoded as `A=0`.
- `A`, encoded as `A=1`.

Qm Source vector register.

RdaHi General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

RdaLo General-purpose register for the low-half of the 64-bit source and destination. This must be an even numbered register.

dt This parameter must be one of the following values:
- `S32`
- `U32`

v See Standard Assembler Syntax Fields

Restrictions

`RdaHi` must not use `SP`.

Post-conditions

There are no condition flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Add across the elements of a vector accumulating the result into a scalar. The 64-bit result is stored across two registers, the upper half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

VADDLV example

```
// 32-bit integer vector intra long addition without accumulation
MOV           R2, #0
MOV           R3, #1000
VIDUP.U32     Q0, R2, #4    // Generates incrementing sequence, starting at 0 with
increments of 4
VMUL.S32      Q0, Q0, Q3    // Multiply by 1000
                            // Q0 =[ 0 4000 8000 12000 ]
VADDLV.S32    R0, R1, Q0    // R0:R1 = sum(Q0[i])     i={0..3}
                            // R0:R1 = 24000

// 32-bit integer vector intra long addition with accumulation
VIDUP.U32     Q1, R2, #4    // Generates incrementing sequence, starting at 16 with
increments of 4
VMUL.S32      Q1, Q1, r3    // Multiply by 1000
                            // Q1 = [ 16000 20000 24000 28000 ]
VADDLVA.S32   R0, R1, Q1    // R0:R1 = R0:R1 + sum(Q1[i])      i={0..3}
                            // R0:R1 (in)  = 24000  (from previous VADDLV.S32 )
                            // R0:R1 (out) = 24000 + 88000 = 112000
```

3.29.11 VADDV

Vector Add Across Vector.

Syntax

```
VADDV{A}<v>.<dt> Rda, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- An unlisted parameter, encoded as `A=0`.
- `A`, encoded as `A=1`.

Qm Source vector register.

Rda General-purpose source and destination register. This must be an even numbered register.

dt This parameter must be one of the following values:
- `S8`
- `U8`
- `S16`
- `U16`
- `S32`
- `U32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Add across the elements of a vector accumulating the result into a scalar. The initial value of the general-purpose destination register can optionally be added to the result.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VADDV example

```
// 32-bit integer vector intra addition without accumulation
MOV           R2, #0
MOV           R3, #1000     // Sequence multiplier
VIDUP.U32     Q0, R2, #4    // Generates 32-bit incrementing sequence,
                            // starting at 0 with increments of 4
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
                            // Q0 =[ 0 4000 8000 12000 ]
VADDV.S32     R0, Q0        // R0 = sum(Q0[i])     i={0..3}
                            // R0 = 24000

// 32-bit integer vector intra addition with accumulation
VIDUP.U32     Q0, R2, #4    // Generates 32-bit incrementing sequence,
                            // starting at 16 (previous sequence continuation)
                            // with increments of 4
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
                            // Q1 = [ 16000 20000 24000 28000 ] (int32x4_t)
VADDVA.S32    R0, Q1        // R0 = R0 + sum(Q1[i])      i={0..3}
                            // R0 (in)  = 24000 (from previous VADDV execution)
                            // R0 (out) = 24000 + 88000 = 112000
```

3.29.12 VCADD

Vector Complex Add with Rotate.

Syntax

```
VCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - I8
  - I16
  - I32

rotate The rotation amount. This parameter must be one of the following values:
- 90 degrees
- 270 degrees

v See Standard Assembler Syntax Fields

Restrictions

When `I32` is used, then `Qd` must not use the same register as `Qm`

Post-conditions

There are no condition flags.

Operation

This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, and a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components of a complex number, respectively. The result is then written to the destination vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VCADD example

```
// 32-bit integer Vector Complex Addition with 2nd vector operand multiplied by +i
MOV           R2, #1
MOV           R3, #1000
VIDUP.U32     Q0, R2, #1    // Generates incrementing sequence,
                            // starting at 1 with increments of 1
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
VIDUP.U32     Q1, R2, #1    // Generates incrementing sequence,
                            // starting at 5 with increments of 1
VMUL.S32      Q1, Q1, R3    // Multiply by 1000
                            // Q0 = [ 1000+2000i 3000+4000i ]
                            //  (is [ 1000 2000 3000 4000 ])
                            // Q1 = [ 5000+6000i 7000+8000i ]
                            //  (is [ 5000 6000 7000 8000 ])
VCADD.I32     Q2,Q0,Q1,#90  // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1)
                            // + i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                            // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1)
                            // + i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                            // Q2 = [ 1000+2000i 3000+4000i ]
                            // + i * [ 5000+6000i 7000+8000i ]
                            //    = [ -5000+7000i -5000+11000i ]
                            //  (is [ -5000 7000 -5000 11000 ])

// 32-bit integer Vector Complex Addition with 2nd vector operand multiplied by -i
                            // Q0 = [ 1000+2000i 3000+4000i ]
                            //  (is [ 1000 2000 3000 4000 ])
                            // Q1 = [ 5000+6000i 7000+8000i ]
                            //  (is [ 5000 6000 7000 8000 ])
VCADD.I32   Q2,Q0,Q1,#270   // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1)
                            // - i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                            // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1)
                            // - i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                            // Q2 = [ 1000+2000i 3000+4000i ]
                            // - i * [ 5000+6000i 7000+8000i ]
                            //    = [ 7000-3000i 11000-3000i ]
                            //   (is [ 7000 -3000 11000 -3000 ])
```

3.29.13 VCADD (floating-point)

Vector Complex Add with Rotate.

Syntax

```
VCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

Parameters

`Qd` Destination vector register.

`Qm` Second source vector register.

`Qn` First source vector register.

`dt`
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - `F16`
  - `F32`

`rotate` The rotation amount. This parameter must be one of the following values:
- `90 degrees`
- `270 degrees`

`v` See Standard Assembler Syntax Fields

Restrictions

When `S32` is used, then `Qd` must not use the same register as `Qm`

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, and a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components of a complex number, respectively. The results are written into the destination vector register.

VCADD (floating-point) example

```
// 32-bit float Vector Complex Addition with 2nd vector operand multiplied by +i
MOV            R2, #1
MOV            R3, #1000
VIDUP.U32      Q0, R2, #1    // Generates incrementing sequence,
                             // starting at 1 with increments of 1
VMUL.S32       Q0, Q0, R3    // Multiply by 1000
VCVT.F32.S32   Q0, Q0        // Convert 32-bit integer into single precision float vector
VIDUP.U32      Q1, R2, #1    // Generates incrementing sequence,
                             // starting at 5 with increments of 1
VMUL.S32       Q1, Q1, R3    // Multiply by 1000
VCVT.F32.S32   Q1, Q1        // Convert 32-bit integer into single precision float vector
                             // Q0 = [ 1000.0+2000.0i 3000.0+4000.0i ]
                             //  (is [ 1000.0 2000.0 3000.0 4000.0 ])
                             // Q1 = [ 5000.0+6000.0i 7000.0+8000.0i ]
                             // (is [ 5000.0 6000.0 7000.0 8000.0 ])
VCADD.F32      q2,q0,q1,#90  // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1)
                             // + i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                             // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1)
                             // + i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                             // Q2 = [ 1000.0+2000.0i 3000.0+4000.0i ]
                             // + i * [ 5000.0+6000.0i 7000.0+8000.0i ]
                             //    = [ -5000.0+7000.0i -5000.0+11000.0i ]
                             //  (is [ -5000.0 7000.0 -5000.0 11000.0 ])

                             // Q0 = [ 1000.0+2000.0i 3000.0+4000.0i ]
                             //  (is [ 1000.0 2000.0 3000.0 4000.0 ])
                             // Q1 = [ 5000.0+6000.0i 7000.0+8000.0i ]
                             //  (is [ 5000.0 6000.0 7000.0 8000.0 ])

// 32-bit float Vector Complex Addition with 2nd vector operand multiplied by -i
VCADD.F32     Q2,Q0,Q1,#270  // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1)
                             // - i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                             // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1)
                             // - i*(Q1[2*i]+i*Q1[2*i+1]))  i={0..1}
                             // Q2 = [ 1000.0+2000.0i 3000.0+4000.0i ]
                             // - i * [ 5000.0+6000.0i 7000.0+8000.0i ]
                             //    = [ 7000.0-3000.0i 11000.0-3000.0i ]
                             //  (is [ 7000.0 -3000.0 11000.0
-3000.0 ])
```

3.29.14 VCLS

Vector Count Leading Sign-bits.

Syntax

```
VCLS<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - S8
  - S16
  - S32

v See Standard Assembler Syntax Fields

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Count the leading sign bits of each element in a vector register.

VCLS example

```
// 32-bit integer incrementing & sign-alternating vector normalization
MOV           R0, #2          // Wrap sequence start
MOV           R1, #4          // Wrap sequence top
MOV           R2, #1
MOV           R3, #100
MOV           R4, #0          // Incrementing sequence start
VIWDUP.U32    Q0, R0, R1, #2  // Alternating {0, 2} sequence
VSUB.S32      q0, Q0, R2      // Alternating {+1, -1} sequence
VMUL.S32      Q0, Q0, R3      // [1 -1 1 -1 ] * 100
VIDUP.U32     Q1, R4, #8      // Generates 32-bit incrementing sequence,
                              // starting at 0 with increments of 8
VMUL.S32      Q0, Q0, Q1      // [100 -100 100 -100 ] .* [0 8 16 24]
MOV           R0, #32         // Maximum sign position for 32-bits elements
                              // Q0 = [ 0x00000000 0xFFFFFCE0 0x00000640 0xFFFFF6A0 ]
                              // Q1 = [ 31 21 20 19 ]
VCLS.S32      Q1, Q0          // Q1[i] = CLS(Q0[i]) i={0..3}
                              // ; vector of leading sign bits

VMINV.S32     R0, Q1          // Get minimum over vector
                              // R0 = 19
VSHL.S32      Q0, R0          // Shift up by 19 (normalize)
                              // After normalization
                              // Q0 = [ 0x00000000 0xE7000000 0x32000000 0xB5000000 ]
```

3.29.15 VCLZ

Vector Count Leading Zeros.

Syntax

```
VCLZ<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - I8
  - I16
  - I32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Count the leading zeros of each element in a vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VCLZ example

```
// 32-bit integer incrementing vector normalization (unsigned)
MOV           R0, #0
MOV           R1, #100
VIDUP.U32     Q0, R0, #8   // Generates 32-bit incrementing sequence, starting at 0 with
increments of 8
VMUL.S32      Q0, Q0, R1   // [0 8 16 24] * 100
MOV           R0, #32      // Maximum sign position for 32-bits elements
                           // Q0 = [ 0x00000000 0x00000320 0x00000640 0x00000960 ]
                           // Q1 = [ 32 22 21 20 ]
VCLZ.S32      Q1, Q0       // Q1[i] = CLZ(Q0[i]) i={0..3} ; vector of leading zero bits count

VMINV.S32     R0, Q1       // Get minimum over vector, R0 = 20
VSHL.S32      Q0, R0       // Normalize
                           // After normalization
                           // Q0 = [ 0x00000000 0x32000000 0x64000000 0x96000000 ]
```

3.29.16 VCMLA (floating-point)

Vector Complex Multiply Accumulate.

Syntax

```
VCMLA<v>.<dt> Qda, Qn, Qm, #<rotate>
```

Parameters

Qda Source and destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - F16
  - F32

rotate The rotation amount. This parameter must be one of the following values:
- 0 degrees
- 90 degrees
- 180 degrees
- 270 degrees

v See Standard Assembler Syntax Fields

Restrictions

- When S32 is used, then Qda must not use the same register as Qm
- When S32 is used, then Qda must not use the same register as Qn

Post-conditions

There are no condition flags.

Operation

This instruction operates on complex numbers that are represented in registers as pairs of elements. Each element holds a floating-point value. The odd element holds the imaginary part of the number, and the even element holds the real part of the number. The instruction performs the computation on the corresponding complex number element pairs from the two source registers and the destination register.

Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.

- If the transformation was a rotation by 0 or 180 degrees, the two elements of the transformed complex number are multiplied by the real element of the first source register.
- If the transformation was a rotation by 90 or 270 degrees, the two elements are multiplied by the imaginary element of the complex number from the first source register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- The result of the multiplication is added on to the existing value in the destination vector register.
- The multiplication and addition operations are fused and the result is not rounded.

VCMLA (floating-point) example

```
// 2 x 16-bit float incrementing sequences Vector Complex Multiply
MOV           R0, #0
VIDUP.U16     Q0, R0, #2       // Incrementing sequence, starting at 0, increment of 2
VCVT.F16.S16  Q0, Q0           // Convert into F16 vector
VIDUP.U16     Q1, R0, #1       // Incrementing sequence, starting at 16, increment of 1
VCVT.F16.S16  Q1, Q1           // Convert into F16 vector
                               // Q0 = [0+2i    4+6i    8+10i   12+14i]
                               // (is [ 0.000 2.000  4.000 6.000  8.000
                               // 10.000  12.000 14.000 ] )
                               // Q1 = [16+17i  18+19i  20+21i  22+23i]
                               // (is [ 16.000 17.000  18.000 19.000
                               // 20.000 21.000  22.000 23.000 ])
VCMUL.F16     Q2, Q0, Q1, #0   // F16 complex mult part 1
                               //  Q2[2*i] = Q0[2*i] * Q1[2*i]
                               //  Q2[2*i+1] = Q0[2*i] * Q1[2*i+1]
VCMLA.F16     Q2, Q0, Q1, #90  // F16 complex multiplication (Q0*Q1) part 2
                               // Q2[2*i] += -Q0[2*i+1] * Q1[2*i+1]
                               // Q2[2*i+1] += Q0[2*i+1] * Q1[2*i]
                               // Q2 = [0+2i    4+6i    8+10i   12+14i]
                               // .* [16+17i  18+19i  20+21i  22+23i]
                               //    = [-34+32i  -42+184i  -50+368i  -58+584i]
                               // ( is [ -34.000 32.000  -42.000 184.000
                               // -50.000 368.000  -58.000 584.000 ] )


// 2 x 16-bit float incrementing sequences Vector Complex Multiply (Conjugate of 1st vector
operand multiplied by 2nd vector)
                               // Q0 = [0+2i    4+6i    8+10i   12+14i]
                               // (is [ 0.000 2.000  4.000 6.000  8.000
                               // 10.000  12.000 14.000 ] )
                               // Q1 = [16+17i  18+19i  20+21i  22+23i]
                               // (is [ 16.000 17.000  18.000 19.000
                               // 20.000 21.000  22.000 23.000 ])
VCMUL.F16     Q2, Q0, Q1, #0   // F16 complex mult part 1
                               //  Q2[2*i] = Q0[2*i] * Q1[2*i]
                               //  Q2[2*i+1] = Q0[2*i] * Q1[2*i+1]
VCMLA.F16     Q2, Q0, Q1, #270  // F16 complex multiplication part 2 (conjugate(Q0) * Q1)
                               // Q2[2*i] += Q0[2*i+1] * Q1[2*i+1]
                               // Q2[2*i+1] += -Q0[2*i+1] * Q1[2*i]
                               // Q2 = conj([0+2i    4+6i    8+10i   12+14i])
                               // .* [16+17i  18+19i  20+21i  22+23i]
                               //    = [34-32i 186-32i 370-32i 586-32i]
                               // (is [34.000 -32.000  186.000 -32.000
                               // 370.000 -32.000  586.000 -32.000])
```

3.29.17 VCMUL (floating-point)

Vector Complex Multiply.

Syntax

```
VCMUL<v>.<dt> Qd, Qn, Qm, #<rotate>
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F16
  - F32

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

`rotate` The rotation amount. This parameter must be one of the following values:

- `0 degrees`
- `90 degrees`
- `180 degrees`
- `270 degrees`

`v` See Standard Assembler Syntax Fields

Restrictions

- When `S32` is used, then `Qd` must not use the same register as `Qm`
- When `S32` is used, then `Qd` must not use the same register as `Qn`

Post-conditions

There are no condition flags.

Operation

This instruction operates on complex numbers that are represented in registers as pairs of elements. Each element holds a floating-point value. The odd element holds the imaginary part of the number, and the even element holds the real part of the number.

The instruction performs the computation on the corresponding complex number element pairs from the two source registers and the destination register.Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.

- If the transformation was a rotation by 0 or 180 degrees, the two elements of the transformed complex number are multiplied by the real element of the first source register.
- If the transformation was a rotation by 90 or 270 degrees, the two elements are multiplied by the imaginary element of the complex number from the first source register.
- The results are written into the destination vector register.

VCMUL (floating-point) example

```
// 2 x 16-bit float incrementing sequences Vector Complex Multiply
MOV            r0, #0
VIDUP.U16      Q0, r0, #2       // Incrementing sequence, starting at 0, increment of 2
VCVT.F16.S16   Q0, Q0           // Convert into F16 vector
VIDUP.U16      Q1, r0, #1       // Incrementing sequence, starting at 16, increment of 1
VCVT.F16.S16   Q1, Q1           // Convert into F16 vector
                                // Q0 = [0+2i    4+6i    8+10i   12+14i]
                                // (is [ 0.000 2.000  4.000 6.000  8.000
                                // 10.000  12.000 14.000 ] )
                                // Q1 = [16+17i  18+19i  20+21i  22+23i]
                                // (is [ 16.000 17.000  18.000 19.000
                                // 20.000 21.000  22.000 23.000 ])
VCMUL.F16      Q2, Q0, Q1, #0   // F16 complex mult part 1
                                //  Q2[2*i] = Q0[2*i] * Q1[2*i]
                                //  Q2[2*i+1] = Q0[2*i] * Q1[2*i+1]
VCMLA.F16      Q2, Q0, Q1, #90  // F16 complex mult part 2
                                // Q2[2*i] += -Q0[2*i+1] * Q1[2*i+1]
                                // Q2[2*i+1] += Q0[2*i+1] * Q1[2*i]
                                // Q2 = [0+2i    4+6i    8+10i   12+14i].* [16+17i  18+19i
20+21i  22+23i]
                                //    = [-34+32i  -42+184i  -50+368i  -58+584i]
                                // ( is [ -34.000 32.000  -42.000
                                // 184.000  -50.000 368.000  -58.000 584.000 ] )


// 2 x 16-bit float incrementing sequences Vector Complex Multiply (Conjugate of 1st vector
operand multiplied by 2nd vector)
                                // Q0 = [0+2i    4+6i    8+10i   12+14i]
                                // (is [ 0.000 2.000  4.000 6.000  8.000
                                // 10.000  12.000 14.000 ] )
                                // Q1 = [16+17i  18+19i  20+21i  22+23i]
                                // (is [ 16.000 17.000  18.000 19.000
                                // 20.000 21.000  22.000 23.000 ])
VCMUL.F16      Q2, Q0, Q1, #0   // F16 complex mult part 1
                                //  Q2[2*i] = Q0[2*i] * Q1[2*i]
                                //  Q2[2*i+1] = Q0[2*i] * Q1[2*i+1]
VCMLA.F16      Q2, Q0, Q1, #270 // F16 complex mult part 2 (conjugate(Q0) * Q1)
                                // Q2[2*i] += Q0[2*i+1] * Q1[2*i+1]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                              // Q2[2*i+1] += -Q0[2*i+1] * Q1[2*i]
                              // Q2 = conj([0+2i    4+6i    8+10i   12+14i])
                              // .* [16+17i  18+19i  20+21i  22+23i]
                              //    = [34-32i 186-32i 370-32i 586-32i]
                              // (is [34.000 -32.000  186.000 -32.000
                              // 370.000 -32.000  586.000 -32.000])

// Individual VCMUL variants operating on 16-bit float
VCMUL.F16     Q2, Q0, Q1, #0   // Q2[2*i] = Q0[2*i]*Q1[2*i]
                               // Q2[2*i+1] = Q0[2*i]*Q1[2*i+1]
                               // Q2 = [ 0.000 0.000  72.000 76.000
                               // 160.000 168.000  264.000 276.000 ]

VCMUL.F16     Q2, Q0, Q1, #180 // Q2[2*i] = -Q0[2*i]*Q1[2*i]
                               // Q2[2*i+1] = -Q0[2*i]*Q1[2*i+1]
                               // Q2 = [ -34.000 32.000  -114.000
                               // 108.000  -210.000 200.000  -322.000 308.000 ]

VCMUL.F16     Q2, Q0, Q1, #90  // Q2[2*i] = -Q0[2*i+1]*Q1[2*i+1]
                               // Q2[2*i+1] = Q0[2*i+1]*Q1[2*i]
                               // Q2 = [ -0.000 -0.000  -72.000 -76.000
                                // -160.000 -168.000  -264.000
-276.000 ]

VCMUL.F16     Q2, Q0, Q1, #270 // Q2[2*i] = Q0[2*i+1]*Q1[2*i+1]
                               // Q2[2*i+1] = -Q0[2*i+1]*Q1[2*i]
                               // Q2 = [ 34.000 -32.000  114.000
                               // -108.000  210.000 -200.000  322.000 -308.000 ]
```

3.29.18 VDDUP, VDWDUP

Vector Decrement and Duplicate, Vector Decrement with Wrap and Duplicate.

Syntax

```
VDDUP<v>.<dt> Qd, Rn, #<imm>
VDWDUP<v>.<dt> Qd, Rn, Rm, #<imm>
```

Parameters

Qd Destination vector register.

Rm Size of the range. Must be a multiple of `imm`. This must be an odd numbered register.

Rn For the VDDUP and VDWDUP variants, this register holds the current offset to start writing into Qd. This must be an even numbered register. Additionally for the VDWDUP variant, Rn must be a multiple of imm.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - U8
  - U16
  - U32

imm The increment between successive element values. This parameter must be one of the following values:
- #1.
- #2.
- #4.
- #8.

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use `SP`.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Creates a vector with elements of successively decrementing values, starting at an offset specified by Rn. The value is decremented by the specified immediate value, which can take the values 1, 2, 4, or 8. For all variants, the updated start offset is written back to Rn. For the wrapping variant, the operation wraps so that the values written to the vector register elements are in the range (0-(Rm-1)). However, if Rn and Rm are not a multiple of imm, or if Rn >= Rm, the operation is CONSTRAINED UNPREDICTABLE, with the resulting values of Rn and Qd UNKNOWN.

VDDUP example

```
// 16-bit vector gather load with decrementing offsets with a step of 2
MOV            R0, #20                // Decrementing sequence start
VDDUP.U16      Q0, R0, #2             // Generator, decrement step of 2
                                      // Q0 = [  20 18  16 14  12 10  8 6  ]
                                      // R0 = 4 after the VDDUP execution
VLDRH.S16      Q1, [R1, Q0, UXTW #1]  // Gather load, base = R1, offset = Q0
                                      // R1 points to int16_t array containing
                                      // {100, 101, 102, 103, 104, ...}
                                      // Offsets are pre-scaled by 2 to
                                      // keep 16-bit alignment
                                      // Q1 = [  120 118  116 114  112 110  108 106  ]
```

VDWDUP example

```
// 16-bit vector decrementing circular load
MOV           R0, #4                  // Decrementing sequence start
MOV           R1, #16                 // Decrementing sequence wrap
VDWDUP.U16    Q0,R0,R1,#1             // Generator, decrement + wrap with a step of 1
                                      // Q0 = [  4 3  2 1  0 15  14 13  ]
                                      // R0 = 12 after VDWDUP execution
VLDRH.S16     Q1, [R2, Q0, UXTW #1]   // Gather load, base = R2, offset = Q0
                                      // R2 points to int16_t array containing
                                      // {100, 101, 102, 103, 104, ...}
                                      // Offsets are pre-scaled by 2 to
                                      // keep 16-bit alignment
                                      // Q1 = [ 104 103  102 101  100 115  114 113  ]

// 2nd iteration
VDWDUP.U16    Q0,R0,R1,#1              // Generator continuation,
                                       // R0 = start at 12, decrement
                                       // + wrap with a step of 1
                                       // Q0 = [  12 11  10 9  8 7  6 5  ]
                                       // R0 = 4 after VDWDUP execution
VLDRH.S16     Q1, [R2, Q0, UXTW #1]    // Gather load, base = r2, offset = Q0
                                       // Q1 = [ 112 111  110 109  108 107 106 105  ]
```

3.29.19 VDUP

Vector Duplicate.

Syntax

```
VDUP<v>.<size> Qd, Rt
```

Parameters

Qd Destination vector register.

Rt Source general-purpose register.

size
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - 32
  - 16
  - 8

v See Standard Assembler Syntax Fields

Restrictions

`Rt` must not use the same register as `SP` and `PC`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

Set each element of a vector register to the value of a general-purpose register.

VDUP example

```
// 8-bit duplication
MVN      R0,#1
VDUP.8   Q0,R0  // Q0[i] = -2  i={0..15}
        // Q0 = [  -2 -2 -2 -2  -2 -2 -2 -2  -2 -2 -2 -2  -2 -2 -2 -2 ]

// 16-bit duplication
MOV      R1, #1234
VDUP.16  Q1,R1  // Q0[i] = 1234  i={0..7}
        // Q1 = [  1234 1234  1234 1234  1234 1234  1234 1234  ]

// 32-bit duplication
MOVW     R2, #0
MOVT     R2, 0x4060  // 3.5f
VDUP.32  Q2,R2  // Q0[i] = 3.5  i={0..3}
        // Q2 = [  3.500 3.500 3.500 3.500  ]
```

3.29.20 VFMA (vector by scalar plus vector, floating-point)

Vector Fused Multiply Accumulate.

Syntax

```
VFMA<v>.<dt> Qda, Qn, Rm
```

Parameters

Qda Accumulator vector register.

Qn Source vector register.

Rm Source general-purpose register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC.

Post-conditions

There are no condition flags.

Operation

Multiply each element in a vector register by a general-purpose register value to produce a vector of results. Each result is then added to its respective element in the destination register. The result of each multiply is not rounded before the addition.

VFMA (vector by scalar plus vector, floating-point) example

```
// Half floating point vector by scalar multiplication + vector
MOV             R0, #0
VIDUP.U16       Q0, R0, #2    // Incrementing sequence,
                              // starting at 0, increment of 2
VCVT.F16.S16    Q0, Q0        // Convert into F16 vector
VIDUP.U16       Q1, R0, #1    // Incrementing sequence,
                              // starting at 16, increment of 1
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVT.F16.S16    Q1, Q1        // Convert into F16 vector
MOV             R0, #0x3800   // R0 = 0.5f15
                              // Q0 (in) =  [ 0.000 2.000  4.000 6.000  8.000
                              // 10.000  12.000 14.000 ]
                              // Q1       = [ 16.000 17.000  18.000 19.000
                              // 20.000 21.000  22.000 23.000 ]

VFMA.F16        Q0,Q1,R0      // Q0[i] = Q0[i] + Q1[i] * R0   i={0..7}
                              // Q0 (out) = [ 8.000 10.500  13.0 15.5
                              // 18.0 20.5 23.0 25.5 ]
```

3.29.21 VFMA, VFMS (floating-point)

Vector Fused Multiply Accumulate, Vector Fused Multiply Subtract.

Syntax

```
VFMA<v>.<dt> Qda, Qn, Qm
VFMS<v>.<dt> Qda, Qn, Qm
```

Parameters

Qda Source and destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Multiply each element of the first source vector register by its respective element in the second vector register. Each result is then added to or subtracted from its respective element in the destination register. The result of each multiply is not rounded before the addition or subtraction.

VFMA (floating-point) example

```
// Sum of square of two incrementing half-precision floating-point vectors
MOV           R0, #0
VIDUP.U16     Q0, R0, #2    // Incrementing sequence, starting at 0, increment of 2
VCVT.F16.S16  Q0, Q0        // Convert into F16 vector
VIDUP.u16     Q1, R0, #1    // incrementing sequence, starting at 16, increment of 1
VCVT.f16.s16  Q1, Q1        // Convert into F16 vector
                            // Q0 = [ 0.000 2.000  4.000 6.000  8.000
                            // 10.000  12.000 14.000 ]
                            // Q1 = [ 16.000 17.000  18.000 19.000
                            //  20.000 21.000  22.000 23.000 ]
VMUL.F16      Q2, Q0, Q0    // Q2[i] = Q0[i] * Q0[i]    i={0..7}
                            // Q0 = [ 0.000 4.000  16.000 36.000
                            // 64.000 100.000  144.000 196.000 ]

VFMA.F16      Q2, Q1, Q1    // Q2[i] += Q1[i] * Q1[i]   i={0..7}
                            // Q2 = [ 256.000 293.000  340.000 397.000
                            // 464.000 541.000  628.000 725.000 ]
```

VFMS (floating-point) example

```
// Difference of square of 2 incrementing single precision FP vectors
MOV           R0, #0
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VIDUP.U32     Q0, r0, #2    // Incrementing sequence, starting at 0, increment of 2
VCVT.F32.S32  Q0, q0        // Convert into F32 vector
VIDUP.U32     Q1, r0, #1    // Incrementing sequence, starting at 8, increment of 1
VCVT.F32.S32  Q1, q1        // Convert into F32 vector
                            // Q0 = [ 0.000 2.000 4.000 6.000 ]
                            // Q1 = [ 8.000 9.000 10.000 11.000 ]
VMUL.F32      Q2, Q0, Q0    // Q2[i] = Q0[i] * Q0[i]    i={0..3}
                            // Q2 = [ 0 4.000 16.000 36.000 ]

VFMS.F32      Q2, Q1, Q1    // Q2[i] -= Q1[i] * Q1[i]   i={0..3}
                            // Q2 = [ -64.000 -77.000 -84.000 -85.000 ]
```

3.29.22 VFMAS (vector by vector plus scalar, floating-point)

Vector Fused Multiply Accumulate Scalar.

Syntax

```
VFMAS<v>.<dt> Qda, Qn, Rm
```

Parameters

Qda Source and destination vector register.

Qn Source vector register.

Rm Source general-purpose register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC.

Post-conditions

There are no condition flags.

Operation

Multiply each element in the source vector by the respective element from the destination vector and add to a scalar value. The resulting values are stored in the destination vector register. The result of each multiply is not rounded before the addition.

VFMAS (vector by vector plus scalar, floating-point) example

```
// Half floating point vector by vector multiplication + scalar
MOV             R0, #0
VIDUP.U16       Q0, R0, #2    // incrementing sequence, starting at 0, increment of 2
VCVT.F16.S16    Q0, Q0        // convert into f16 vector
VIDUP.U16       Q1, R0, #1    // incrementing sequence, starting at 16, increment of 1
VCVT.F16.S16    Q1, Q1        // convert into f16 vector
MOV             R0, #0x3800   //  R0 = 0.5f15
                              // Q0 (in) = [ 0.000 2.000  4.000 6.000
                              // 8.000 10.000  12.000 14.000 ]
                              // Q1 = [ 16.000 17.000  18.000 19.000
                              // 20.000 21.000  22.000 23.000 ]

VFMAS.F16       Q0, Q1, R0    // Q0[i] = R0 + Q0[i] * Q1[i]   i={0..7}
                              // Q0 (out) = [ 0.500 34.500  72.500 114.500
                              // 160.500 210.500  264.500 322.500 ]
```

3.29.23 VHADD

Vector Halving Add.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VHADD<v>.<dt> Qd, Qn, Rm
VHADD<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the size of the elements in the vector.
- This parameter determines the following values:
  - S8
  - U8
  - S16
  - U16
  - S32
  - U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC.

Post-conditions

There are no condition flags.

Operation

Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is halved before being written to the destination vector register.

VHADD example

```
// Vector Adding with halving of 2 x 16-bit integer vectors
VLDRH.S16      Q0, [r0], #16    // Contiguous 16-bit vector load (+ post-increment)
VLDRH.S16      Q1, [r0], #16    // Contiguous 16-bit vector load (+ post-increment)
                                // Q0 = [  -32768 12288  -20480 28672
                                // 28672 -20480  12288 32767  ]
                                // Q1 = [  4097 32767  12289 16385  20481
                                // 24577  28673 -32768  ]
VHADD.S16      Q2, Q0, Q1       // Q2[i] = (Q0[i] + Q1[i])/2  i={0..7}
                                // Q2 = [  -14336 22527  -4096 22528
                                // 24576 2048  20480 -1  ]

// Vector Adding with halving of a 16-bit integer vector with a scalar
MOV            R0, #500
                              // Q0 = [  -32768 12288  -20480 28672
                              // 28672 -20480  12288 32767  ]
VHADD.S16      Q2, Q0, R0     // Q2[i] = (Q0[i] + R0)/2     i={0..7}
                              // Q2 = [  -16134  6394  -9990  14586
                              // 14586 -9990  6394  16633  ]

// Vector Adding & Subtracting with halving of 2 x 32-bit integer vectors
VLDRW.S32        Q0, [R0]       // Vector load in Q0, base address in R0
VLDRW.S32        Q1, [R1]       // Vector load in Q1, base address in R1

VHADD.S32        Q2, Q0, Q1     // Q2[i] = (Q0[i] + Q1[i])/2  i={0..3}
VHSUB.S32        Q3, Q0, Q1     // Q2[i] = (Q0[i] - Q1[i])/2  i={0..3}
VSTRW.S32        Q0, [R0], #16  // Store Q0, base address in R0, post-increment
VSTRW.S32        Q1, [R1], #16  // Store Q1, base address in R1, post-increment
```

3.29.24 VHCADD

Vector Halving Complex Add with Rotate.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VHCADD<v>.<dt> Qd, Qn, Qm, #<rotate>
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - S8
  - S16
  - S32

rotate The rotation amount. This parameter must be one of the following values:
- 90 degrees.
- 270 degrees.

v See Standard Assembler Syntax Fields

Restrictions

When I32 is used, then Qd must not use the same register as Qm

Post-conditions

There are no condition flags.

Operation

This instruction performs a complex addition of the first operand with the second operand rotated in the complex plane by the specified amount. A 90 degree rotation of this operand corresponds to a multiplication by a positive imaginary unit, while a 270 degree rotation corresponds to a multiplication by a negative imaginary unit. Even and odd elements of the source vectors are interpreted to be the real and imaginary components, respectively, of a complex number. The result is halved before being written to the destination register.

VHCADD example

```
// 32-bit integer Vector Halving Complex Addition with 2nd vector
//  operand multiplied by +i
MOV           R2, #1
MOV           R3, #1000
VIDUP.U32     Q0, R2, #1       // Generates incrementing sequence,
                               // starting at 1 with increments of 1
VMUL.S32      Q0, Q0, R3       // Multiply by 1000
VIDUP.U32     Q1, R2, #1       // Generates incrementing sequence,
                               // starting at 5 with increments of 1
VMUL.S32      Q1, Q1, R3       // Multiply by 1000
                               // Q0 = [ 1000+2000i 3000+4000i ]
                               // [ 1000 2000 3000 4000 ]
                               // Q1 = [ 5000+6000i 7000+8000i ]
                               // [ 5000 6000 7000 8000 ]
VHCADD.S32    Q2, Q0, Q1, #90  // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1]
                               // + i*(Q1[2*i]+i*Q1[2*i+1]))/2  i={0..1}
                               // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1]
                               // + i*(Q1[2*i]+i*Q1[2*i+1]))/2  i={0..1}
                               // Q2 = ([ 1000+2000i 3000+4000i ]
                               // + i * [ 5000+6000i 7000+8000i ])/2
                               //    = [ -2500 3500 -2500 5500 ]

// 32-bit integer Vector Halving Complex Addition with 2nd vector
// operand multiplied by +i
                               // Q0 = [ 1000+2000i 3000+4000i ]
                               // [ 1000 2000 3000 4000 ]
                               // Q1 = [ 5000+6000i 7000+8000i ]
                                // [ 5000 6000 7000 8000 ]
VHCADD.S32    Q2, Q0, Q1, #270 // Q2[2*i]  = Real(Q0[2*i]+i*Q0[2*i+1]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                              // - i*(Q1[2*i]+i*Q1[2*i+1]))/2  i={0..1}
                              // Q2[2*i+1]= Imag(Q0[2*i]+i*Q0[2*i+1]
                              // - i*(Q1[2*i]+i*Q1[2*i+1]))/2  i={0..1}
                              // Q2 = ([ 1000+2000i 3000+4000i ]
                              // + i * [ 5000+6000i 7000+8000i ])/2
                              //    = [ 3500 -1500 5500 -1500 ]
```

3.29.25 VHSUB

Vector Halving Subtract.

Syntax

```
VHSUB<v>.<dt> Qd, Qn, Qm
VHSUB<v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the size of the elements in the vector.
- This parameter determines the following values:
  - S8
  - U8
  - S16
  - U16
  - S32
  - U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC.

Post-conditions

There are no condition flags.

Operation

Subtract the value of the elements in the second source vector register from the respective elements in the first source vector register. The result is halved before being written to the destination vector register.

VHSUB example

```
// Vector Subtract with halving of 2 x 32-bit integer vectors
VLDRH.S16      Q0, [R0], #16  // Contiguous 16-bit vector load (+ post-increment)
VLDRH.S16      Q1, [R0], #16  // Contiguous 16-bit vector load (+ post-increment)
                              // Q0 = [ 1000 2000 3000 4000 ]
                              // Q1 = [ 5000 9000 13000 17000 ]
VHSUB.S32      Q2, Q0, Q1     // Q2[i] = (Q1[0] - Q1[i])/2   i={0..3}
                              // Q2 = [ -2000 -3500 -5000 -6500 ]

// Vector Subtract with halving of a 32-bit integer vector and a scalar
MOV         R0, 500
                              // Q0 = [ 1000 2000 3000 4000 ]
VHSUB.S32   Q2, Q0, R0        // Q2[i] = (Q1[0] - 500)/2   i={0..3}
                              // Q2 = [ 250 750 1250 1750 ]

// Vector Subtract with halving of 2 x 32-bit integer vectors
// highlights overflow prevention if intermediate value exceeds signed 32-bit max. magnitude
MOV         R0, #0x81000000  //  -0.9921875 in Q.31
MOV         R1, #0x7f000000  //  0.9921875 in Q.31
VMOV.S32    Q0[0], R0
VMOV.S32    Q1[0], r1
                            // Q0 = [ -2130706432 2000 3000 4000 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                              // Q1 = [ 2130706432 9000 13000 17000 ]
VHSUB.S32   Q2, Q0, Q1        // Q2[i] = (Q1[0] - Q1[i])/2   i={0..3}
                              // Q2 = [ -2130706432 -3500 -5000 -6500 ]
```

3.29.26 VIDUP, VIWDUP

Vector Increment and Duplicate, Vector Increment with Wrap and Duplicate.

Syntax

```
VIDUP<v>.<dt> Qd, Rn, #<imm>
VIWDUP<v>.<dt> Qd, Rn, Rm, #<imm>
```

Parameters

`Qd` Destination vector register.

`Rm` Size of the range. Must be a multiple of `imm`. This must be an odd numbered register.

`Rn` For the VIDUP and VIWDUP variants, this register holds the current offset to start writing into Qd. This must be an even numbered register. Additionally for the VIWDUP variant, Rn must be a multiple of imm.

`dt`
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - `U8`
  - `U16`
  - `U32`

`imm` The increment between successive element values. This parameter must be one of the following values:
- `#1`
- `#2`
- `#4`
- `#8`

`v` See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use `SP`.

Post-conditions

There are no condition flags.

Operation

Creates a vector with elements of successively incrementing values, starting at an offset specified by Rn. The value is incremented by the specified immediate value, which can take the values 1, 2, 4, or 8. For all variants, the updated start offset is written back to Rn. For the wrapping variant, the operation wraps so that the values written to the vector register elements are in the range 0-Rm. However, if Rn and Rm are not a multiple of imm, or if Rn >= Rm, the operation is CONSTRAINED UNPREDICTABLE, with the resulting values of Rn and Qd UNKNOWN.

VIDUP, VIWDUP example

```
```

3.29.27 VMLA (vector by scalar plus vector)

Vector Multiply Accumulate.

Syntax

```
VMLA<v>.<dt> Qda, Qn, Rm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qda Accumulator vector register.

Qn Source vector register.

Rm Source general-purpose register.

dt This parameter determines the following values:
- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Multiply each element in the source vector by a scalar value and add to the respective element from the destination vector. Store the result in the destination register.

VMLA (vector by scalar plus vector) example

```
// 32-bit integer vector by scalar plus vector
MOV           R2, #0
MOV           R3, #1000
VIDUP.u32     Q0, R2, #1    // generates incrementing sequence, starting at 0 with step of 1
VMUL.s32      Q0, Q0, R3    // multiply by 1000
VDDUP.u32     Q1, R2, #1    // generates decrementing sequence, starting at 4 with step of 1
VMUL.s32      Q1, Q1, R3    // multiply by 1000
MOV           R0, #1000     // scalar multiplier
                            // Q0 = [ 0 1000 2000 3000 ]
                            // Q1 = [ 4000 3000 2000 1000 ]

VMLA.s32      Q0, Q1, R0    // Q0[i] = Q0[i] + Q1[i] * R0 i={0..3}
                            // Q0 result = [ 4000000 3001000 2002000 1003000 ]
```

3.29.28 VMLADAV

Vector Multiply Add Dual Accumulate Across Vector.

Syntax

```
VMLADAV{A}{X}<v>.<dt> Rda, Qn, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- A.

Qm Second source vector register.

Qn First source vector register.

Rda General-purpose source and destination register. This must be an even numbered register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated and the lower 32 bits written back to the general-purpose destination register. The initial value of the general-purpose destination register can optionally be added to the result.

VMLADAV example

```
// 16-bit Vector Multiply Add Dual Accumulate Across Vector
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3    // Multiply by 1000
VDDUP.U16     Q1, R2, #1    // Generates decrementing sequence, starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3    // Multiply by 1000

// non-accumulated
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLADAV.S16   R0, Q0, Q1    // R0 = sum(Q0[i] * Q1[i])
                            // R0 = 84000000

// with accumulation
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLADAVA.S16   R0, Q0, Q1   // R0 = R0 + sum(Q0[i] * Q1[i])
                            // R0 input = 84000000
                            // R0 output = 168000000

// with accumulation and exchange
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLADAVX.S16   R0, Q0, Q1   // R0 = sum(Q0[2*i] * Q1[2*i+1] + Q0[2*i+1] * Q1[2*i])
                            // R0 = 88000000
```

3.29.29 VMLALDAV

Vector Multiply Add Long Dual Accumulate Across Vector.

Syntax

```
VMLALDAV{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

`A` Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- `A`.

`Qm` Second source vector register.

`Qn` First source vector register.

`RdaHi` General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

`RdaLo` General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register.

`X` Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- `X`.

`dt` This parameter determines the following values:
- `S8`
- `U8`
- `S16`
- `U16`
- `S32`
- `U32`

`v` See Standard Assembler Syntax Fields

Restrictions

`RdaHi` must not use `SP`

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated. The 64 bit result is stored across two registers, the upper-half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

VMLALDAV example

```
// 16-bit Vector Multiply Add Long Dual Accumulate Across Vector
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1       // generates incrementing sequence, starting at 0 with step
of 1
VMUL.S16      Q0, Q0, R3       // multiply by 1000
VDDUP.U16     Q1, R2, #1       // generates decrementing sequence, starting at 8 with step
of 1
VMUL.S16      Q1, Q1, R3       // multiply by 1000

// non-accumulated
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLALDAV.S16   R0, R1, Q0, Q1  // R0:R1 = sum(Q0[i] * Q1[i])
                               // R0:R1 = 84000000

// with accumulation
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLALDAVA.S16   R0, R1, Q0, Q1 // R0:R1 = R0:R1 + sum(Q0[i] * Q1[i])
                               // R0:R1 input = 84000000
                               // R0:R1 output = 168000000

// with accumulation and exchange
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLALDAVX.S16   R0, R1, Q0, Q1 // R0:R1 = sum(Q0[2*i] * Q1[2*i+1] + Q0[2*i+1] * Q1[2*i])
                               // R0:R1 = 88000000
```

3.29.30 VMLALV

Vector Multiply Accumulate Long Across Vector.

Syntax

```
VMLALV{A}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

Parameters

None.

Restrictions

`RdaHi` must not use `SP`

Post-conditions

There are no condition flags.

Operation

This is an alias of VMLALDAV without exchange.

VMLALV example

```
// 16-bit Vector Multiply Accumulate Long Across Vector
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1       // Generates incrementing sequence,
                               // starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3       // Multiply by 1000
VDDUP.U16     Q1, R2, #1       // Generates decrementing sequence,
                                // starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3       // Multiply by 1000
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]

VMLALV.S16   R0, R1, Q0, Q1    // R0:R1 = sum(Q0[i] * q1[i]) i={0..7}
                               // R0:R1 = 84000000
```

3.29.31 VMLAS (vector by vector plus scalar)

Vector Multiply Accumulate Scalar.

Syntax

```
VMLAS<v>.<dt> Qda, Qn, Rm
```

Parameters

`Qda` Source and destination vector register.

`Qn` Source vector register.

`Rm` Source general-purpose register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Multiply each element in the source vector by the respective element from the destination vector and add to a scalar value. Store the result in the destination register.

VMLAS (vector by vector plus scalar) example

```
// 32-bit integer vector by vector plus scalar
MOV           R2, #0
MOV           R3, #1000
VIDUP.U32     Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
VDDUP.U32     Q1, R2, #1    // Generates decrementing sequence, starting at 4 with step of 1
VMUL.S32      Q1, Q1, R3    // Multiply by 1000
MOV           R0, #1000     // Scalar accumulator
                            // Q0 = [ 0 1000 2000 3000 ] (input)
                            // Q1 = [ 4000 3000 2000 1000 ]

VMLAS.S32     Q0, Q1, R0    // Q0[i] = R0 + Q0[i] * Q1[i] i={0..3}
                            // Q0 = [  1000 3001000 4001000 3001000 ] (result)
```

3.29.32 VMLAV

Vector Multiply Accumulate Across Vector.

Syntax

```
VMLAV{A}<v>.<dt> Rda, Qn, Qm
```

Parameters

None.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

This is an alias of VMLADAV without exchange.

VMLAV example

```
// 16-bit integer Vector Multiply Accumulate Across Vector
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1       // Generates incrementing sequence,
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                                    // starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3            // Multiply by 1000
VDDUP.U16     Q1, R2, #1            // Generates decrementing sequence,
                                    // starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3            // Multiply by 1000
                                    // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                                    // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]

VMLAV.S16     R0, Q0, Q1            // R0 = sum(Q0[i] * Q1[i])
                                    // R0 = 84000000
```

3.29.33 VMLSDAV

Vector Multiply Subtract Dual Accumulate Across Vector.

Syntax

```
VMLSDAV{A}{X}<v>.S8 Rda, Qn, Qm
VMLSDAV{A}{X}<v>.<dt> Rda, Qn, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- A.

Qm Second source vector register.

Qn First source vector register.

Rda General-purpose source and destination register. This must be an even numbered register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  — S16
  — S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated and the lower 32 bits written back to the general-purpose destination register. The initial value of the general-purpose destination register can optionally be added to the result.

VMLSDAV example

```
// 16-bit integer Vector Multiply Subtract Dual Accumulate Across Vector.
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1            // Generates incrementing sequence,
                                    // starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3            // Multiply by 1000
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VDDUP.U16     Q1, R2, #1       // Generates decrementing sequence,
                               // starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3       // Multiply by 1000

// Non-accumlated
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLSDAV.S16   R0, Q0, Q1       // R0 = sum(Q0[2*i] * Q1[2*i] - Q0[2*i+1] * Q1[2*i+1])
                               // R0 = -4000000

// With accumulation
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLSDAVA.S16   R0, Q0, Q1      // R0 = R0 - sum(Q0[2*i] * Q1[2*i] - Q0[2*i+1] * Q1[2*i+1])
                               // R0 input = -4000000
                               // R0 output = -8000000

// With accumulation and exchange
                               // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                               // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMLSDAVX.S16   R0, Q0, Q1      // r0 = sum(Q0[2*i+1] * Q1[2*i] - Q0[2*i] * Q1[2*i+1])
                               // R0 = 32000000
```

3.29.34 VMLSLDAV

Vector Multiply Subtract Long Dual Accumulate Across Vector.

Syntax

```
VMLSLDAV{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- A.

Qm Second source vector register.

Qn First source vector register.

RdaHi General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

RdaLo General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - S16
  - S32

v See Standard Assembler Syntax Fields

Restrictions

RdaHi must not use SP

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated. The 64 bit result is stored across two registers, the upper-half is stored in an odd-numbered register and the lower half is stored in an even-numbered register. The initial value of the general-purpose destination registers can optionally be added to the result.

VMLSLDAV example

```
// 32-bit integer Vector Multiply Subtract Long Dual Accumulate Across Vector
MOV            R2, #0
MOV            R3, #1000
VIDUP.U32      Q0, R2, #1       // Generates incrementing sequence,
                                // starting at 0 with step of 1
VMUL.S32       Q0, Q0, R3       // Multiply by 1000
VDDUP.U32      Q1, R2, #1       // Generates decrementing sequence,
                                // starting at 4 with step of 1
VMUL.S32       Q1, Q1, R3       // Multiply by 1000

// non-accumulated
                                // Q0 = [ 0 1000 2000 3000 ]
                                //   % [0 + i*1000 2000+i*3000]
                                // Q1 = [ 4000 3000 2000 1000 ]
                                //   % [4000 + i*3000 2000 + i*10000]
VMLSLDAV.S32   R0, R1, Q0, Q1   // 32-bit integer complex dot product,
                                // real part, no accumulation
                                // R0:R1 = Real(cmplxDot(Q1, Q2))
                                // = -2000000

// accumulated version
                                // Q0 = [ 0 1000 2000 3000 ]
                                //     % [0 + i*1000 2000+i*3000]
                                // Q1 = [ 4000 3000 2000 1000 ]
                                //  % [4000 + i*3000 2000 + i*10000]
VMLSLDAVA.S32  R0, R1, Q0, Q1   // 32-bit integer complex dot product,
                                 // real part with accumulation
                                // = -4000000
```

3.29.35 VMUL

Vector Multiply.

Syntax

```
VMUL<v>.<dt> Qd, Qn, Rm
VMUL<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values:
  - I8
  - I16
  - I32

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Multiply the value of the elements in the first source vector register by either the respective elements in the second source vector register or a general-purpose register. The result is then written to the destination vector register.

VMUL example

```
// 8-bit integer vector multiply
MOV           R0, #0
MOV           R1, #10
VIDUP.U8      Q0, R0, #1    // 8-bit incrementing sequence starting at 0 with a step of 1
VIDUP.U8      Q1, R0, #1    // 8-bit incrementing sequence starting at 16 with a step of 1
                            // Q0 = [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ]
                            // Q1 = [16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 ]

VMUL.U8       Q2, Q0, Q1    // Q2[i] = Q0[i] * q1[i]     i={0..15}
                            // Q2 = [0 17 36 57 80 105 132 161 192 225 4 41 80 121 164 209 ]
(unsaturated)

// Vector by register variant
                            // Q0 = [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ]
VMUL.U8       Q2, Q0, R1    // Q2[i] = Q0[i] * r1        i={0..15}
                            // Q2 = [0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150]
```

3.29.36 VMUL (floating-point)

Vector Multiply.

Syntax

```
VMUL<v>.<dt> Qd, Qn, Rm
VMUL<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt
- Indicates the floating-point format used.
- This parameter must be one of the following values:
  - F32
  - F16

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

There are no condition flags.

Operation

Multiply the value of the elements in the first source vector register by either the respective elements in the second source vector register or a general-purpose register.

VMUL (floating-point) example

```
// Multiplication of a 16-bit float vector with a scalar
MOV           R2, #1
MOV           R0, #0            // Wrap sequence bottom
MOV           R1, #4            // Wrap sequence top
VIWDUP.U16    Q0, R0, R1, #2    // Alternating {0, 2} sequence
VSUB.S16      Q0, Q0, R2        // Alternating {-1, +1} sequence
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VCVY.F16.S16  Q0, Q0            // Convert 16-bit integer to half precision float vector
MOVW          R0, #0x5100       // R0 = 40.0f16 offset
                                // Q0 = [ -1.000 1.000  -1.000 1.000
                                // -1.000 1.000  -1.000 1.000 ]

VMUL.F16      Q2, Q0, R0       // Q1[i] = Q0[i] * R0   i={0..7}
                               // Q2 = [ -40.000 40.000  -40.000 40.000
                               // -40.000 40.000  -40.000 40.000  ]

// Multiplication of 2 x 16-bit float vectors
VIDUP.U16     Q1, R2, #4      // Generates incrementing sequence,
                              // starting at 1 with increments of 4
VCVT.F16.S16  Q1, Q1          // Convert 16-bit integer to half precision float vector
                              // Q0 = [ -1.000 1.000  -1.000 1.000
                              // -1.000 1.000  -1.000 1.000 ]
                              // Q1 = [ 1.000 5.000  9.000 13.000
                              // 17.000 21.000  25.000 29.000 ]

VMUL.F16      Q2, Q0, Q1      // Q2[i] = Q0[i] * Q1[i]   i={0..7}
                              // Q2 = [ -1.000 5.000  -9.000 13.000
                              // -17.000 21.000  -25.000 29.000 ]
```

3.29.37 VMULH, VRMULH

Vector Multiply Returning High Half, Vector Rounding Multiply Returning High Half.

Syntax

```
VMULH<v>.<dt> Qd, Qn, Qm
VRMULH<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Multiply each element in a vector register by its respective element in another vector register and return the high half of the result. The result is optionally rounded before the high half is selected.

VMULH example

```
// 16-bit integer Vector Multiply Returning High Half
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1    // Generates incrementing sequence,
                            // starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3    // Multiply by 1000
VDDUP.U16     Q1, R2, #1    // Generates decrementing sequence,
                            // starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3    // Multiply by 1000
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]

VMULH.S16     Q2, Q0, Q1    // Q2[i] = (Q0[i] * Q1[i]) >> 16 i={0..7}
                            // Q2 = [  0 106 183 228 244 228 183 106 ]
```

VRMULH example

```
// 16-bit integer Vector Rounding Multiply Returning High Half
VLDRH.16             Q0, [R0]   // 16-bit contiguous load, R0 points to bufer containing
                                // {32767 8193  12289 16385...}
VLDRH.16             Q1, [R1]   // 16-bit contiguous load, R1 points to bufer containing
                                // {32767 12288  -20480 28672...}
                                // Q0 = [  32767 8193  12289 16385  20481
                                // 24577  28673 -32768  ]
                                // Q1 = [  32767 12288  -20480 28672
                                // 28672 -20480  12288 -32768  ]

VRMULH.S16           Q2, Q0, Q1 // Q2[i] = ((Q0[i] * Q1[i]) + (1<<15)) >> 16     i={0..7}
                                // Q2 = [  16383 1536  -3840 7168
                                // 8960 -7680  5376 16384  ]
```

3.29.38 VMULL (integer)

Vector Multiply Long.

Syntax

```
VMULL<T><v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

T Specifies which half of the source element is used. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt This parameter determines the following values:
- — `S8`
  — `U8`
  — `S16`
  — `U16`
  — `S32`
  — `U32`

v See Standard Assembler Syntax Fields

Restrictions

- When `I32` is used, then `Qd` must not use the same register as `Qm`
- When `I32` is used, then `Qd` must not use the same register as `Qn`

Post-conditions

There are no condition flags.

Operation

Performs an element-wise integer multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements. The operation produces a double-width result.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMULL (integer) example

```
// 16-bit integer Vector Multiply Long (32-bit integer vector output)
MOV           R2, #0
MOV           R3, #1000
VIDUP.U16     Q0, R2, #1    // Generates incrementing sequence,
                            // starting at 0 with step of 1
VMUL.S16      Q0, Q0, R3    // Multiply by 1000
VDDUP.U16     Q1, R2, #1    // Generates decrementing sequence,
                            // starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3    // Multiply by 1000

// Multiply bottom parts (of adjacent pairs source)
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMULLB.S16     Q2, Q0, Q1   // Q2[i] = Q0[2*i] * Q1[2*i]     i={0..3}
                            // Q2 = [ 0 12000000 16000000 12000000 ]

// Multiply top parts (of adjacent pairs source)
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMULLT.S16     Q3, Q0, Q1   // Q3[i] = Q0[2*i+1] * Q1[2*i+1]  i={0..3}
                            // Q3 = [ 7000000 15000000 15000000 7000000 ]
```

3.29.39 VMULL (polynomial)

Vector Multiply Long.

Syntax

```
VMULL<T><v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

T Specifies which half of the source element is used. This parameter must be one of the following values:
- `B`, indicates bottom half.
- `T`, indicates top half.

dt Specifies whether to do 8x8-16 or 16x16-32 polynomial multiplications. This parameter must be one of the following values:
- `P8`, indicates 8x8-16
- `P16`, indicates 16x16-32.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Performs an element-wise polynomial multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements. The operation produces a double-width result.

VMULL (polynomial) example

```
// 16-bit integer Vector element-wise polynomial multiplication
MOV           R2, #0
MOV           R3, #1000
VIDUP.u16     Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VMUL.S16      Q0, Q0, R3    // Multiply by 1000
VDDUp.U16     Q1, R2, #1    // Generates decrementing sequence, starting at 8 with step of 1
VMUL.S16      Q1, Q1, R3    // Multiply by 1000

// multiply bottom parts (of adjacent pairs source)
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMULLB.P16     Q2, Q0, Q1   // Q2[i] = PolynomialMult(Q0[2*i], Q1[2*i])
                            // Q2 = [ 0 7029504 5587968 7029504 ]

// multiply top parts (of adjacent pairs source)
                            // Q0 = [  0 1000  2000 3000  4000 5000  6000 7000  ]
                            // Q1 = [  8000 7000  6000 5000  4000 3000  2000 1000  ]
VMULLT.P16     Q3, Q0, Q1   // Q3[i] = PolynomialMult(q0[2*i+1], q1[2*i+1])
                            // Q3 = [ 2433984 10668480 10668480 2433984 ]
// Note :
// 1000 = x^3 + x^5 + x^6 + x^7 + x^8 + x^9
// 7000 = x^3 + x^4 + x^6 + x^8 + x^9 + x^11 + x^12
// 1000*7000 = (x^3 + x^5 + x^6 + x^7 + x^8 + x^9)
// *
// (x^3 + x^4 + x^6 + x^8 + x^9 + x^11 + x^12)
// =
// x^6 + x^7 + x^8 + 3 x^9 + 2 x^10 + 4 x^11 + 4 x^12 + 3 x^13 +
//  4 x^14 + 4 x^15 + 3 x^16 + 4 x^17 + 3 x^18 + 2 x^19 + 2 x^20 + x^21
// =
// 1 x^6 + 1 x^7 + 1 x^8 + 1 x^9 + 0 x^10 + 0 x^11 + 0 x^12 + 1 x^13 +
//  0 x^14 + 0 x^15 + 1 x^16 + 0 x^17 + 1 x^18 + 0 x^19 + 0 x^20 + x^21 (modulo-2)
// = 0x2523C0 = 2433984
```

3.29.40 VNEG

Vector Negate.

Syntax

```
VNEG<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Negate the value of each element in a vector register.

VNEG example

```
// 32-bit integer vector negate
MOV           R0, #0
VIDUP.U32     Q0, R0, #2     // generates incrementing sequence, starting at 0 with step of 2
                             // Q0 = [ 0 2 4 6 ]

VENG.S32      Q1, Q0         // Q1[i] = -Q0[i]    i={0..3}
                             // Q1 = [ 0 -2 -4 -6 ]
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.29.41 VNEG (floating-point)

Vector Negate.

Syntax

```
VNEG<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt Indicates the floating-point format used. This parameter must be one of the following values:
- F16
- F32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Negate the value of each element of a vector register.

VNEG (floating-point) example

```
// 32-bit float vector negate
MOV            R0, #0
VIDUP.U32      Q0, R0, #2     // Generates incrementing sequence, starting at 0 with step of 2
VCVT.F32.S32   Q0, Q0         // Convert 32-bit integer to single precision float vector
                              // Q0 = [ 0.0 2.0 4.0 6.0 ]

VNEG.F32         Q1, Q0       // Q1[i] = -Q0[i]    i={0..3}
                              // Q1 = [ 0.0 -2.0 -4.0 -6.0 ]
```

3.29.42 VQABS

Vector Saturating Absolute.

Syntax

```
VQABS<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values
- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

Updates QC flag in FPSCR register.

Operation

Compute the absolute value of and saturate each element in a vector register.

VQABS example

```
// 16-bit integer Vector Saturating Absolute
VLDRH.16        Q0, [R0]    // Contiguous vector load
                            // Q0 = [  32767 1000  -2000 3000  -4000 5000  -6000 -32768  ]

VQABS.S16       Q1, Q0      // Q1[i] = SSAT16(|Q0[i]|)    i={0..7}
                            // Q1 = [  32767 1000  2000 3000  4000 5000  6000 32767  ]
```

3.29.43 VQADD

Vector Saturating Add.

Syntax

```
VQADD<v>.<dt> Qd, Qn, Qm
VQADD<v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

Updates QC flag in FPSCR register.

Operation

Add the value of the elements in the first source vector register to either the respective elements in the second source vector register or a general-purpose register. The result is saturated before being written to the destination vector register.

VQADD example

```
// 2 x 8-bit vector saturating add
MOV             R0, #0
VIDUP.U8        Q0, R0, #4   // Generates 8-bit incrementing sequence,
                             // starting at 0 with increments of 4
VIDUP.U8        Q1, R0, #4   // Generates 8-bit incrementing sequence,
                             // starting at 16 with increments of 4
                             // Q0 = [0 4 8 12 .. 60]
                             // Q1 = [64 68 72 .. 124]
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VQADD.S8        Q2, Q0, Q1   // Q0[i] = SSAT8(Q0[i] + Q1[i])    i={0..15}
                             // Q2 = [ 64 72 80 88 96 104 112 120 127 127 ... 127]

// 32-bit vector saturating add with scalar
MOV             R0, #0x40000000

VLDRW.32        Q0, [R1]    // Contiguous 32-bit vector load
                            // Q0 = [ 0 536870912 1073741824 1610612736 ]

VQADD.S32       Q2, Q0, R0  // Q2[i] = SSAT32(Q0[i] + R0)     i={0..3}
                            // Q2 = [ 1073741824 1610612736 2147483647 2147483647 ]
```

3.29.44 VQDMLADH, VQRDMLADH

Vector Saturating Doubling Multiply Add Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Add Dual Returning High Half.

Syntax

```
VQRDMLADH{X}<v>.<dt> Qd, Qn, Qm
VQDMLADH{X}<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values
- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together and doubling the result. The higher halves of the resulting values are selected as the final results. The base variant writes the results into the lower element of each pair of elements in the destination register, whereas the exchange variant writes to the upper element in each pair. The results are optionally rounded before the higher half is selected and saturated.

VQDMLADH example

```
// 32-bit integer Vector Saturating Doubling Multiply Add Dual Returning High Half
VLDRW.32        Q0, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q1, [r1]    // 32-bit contiguous vector load
                            // Q0 = [ 268435456 536870912 805306368 1073741824 ]
                            // Q1 = [ -268435456 805306368 -1342177280 1879048192 ]
VMOV.S32        Q2, #0
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
VQDMLADH.S32    Q2, Q0, Q1  // Real parts of FXMULT(Q0, conjugate(Q1)),
                            // imag parts are untouched
                            // Q2[2*i] = Q0[2*i]*Q1[2*i] + Q0[2*i+1]*Q1[2*i+1]   i={0..1}
                            // Q2 =[ 167772160 0 436207616 0 ]

// exchange version
VMOV.S32        Q3, #0
                            // Q0 = [ 268435456 536870912 805306368 1073741824 ]
                            // Q1 = [ -268435456 805306368 -1342177280 1879048192 ]
VQDMLADHX.S32   Q3, Q0, Q1  // Imaginary parts of FXMULT(Q0, Q1),
                            // real parts are untouched
                            // Q3[2*i+1] = Q0[2*i+1]*Q1[2*i] + Q0[2*i]*Q1[2*i+1]
                            // Q3 = [ 0 33554432 0 33554432 ]

// FXMULT=saturated multiplication with doubling and high part extraction
```

VQRDMLADH example

```
// 32-bit integer Vector Saturating Rounding Doubling (Multiply Add Dual Returning High Half)
VLDRW.32        Q0, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q1, [R1]    // 32-bit contiguous vector load
                            // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                            // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]
VMOV.S32        Q2, #0
VQRDMLADH.S32   Q2, Q0, Q1  // Real parts of FXMULTR(Q0, conj(Q1)),
                            // imaginary parts are untouched
                            // Q2[2*i] = Q0[2*i]*Q1[2*i] + Q0[2*i+1]*Q1[2*i+1]   i={0..1}
                            // Q2 = [ 167772161 0 436207617 0 ]
// Exchange variant
VMOV.S32        Q3, #0
                            // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                            // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]
VQRDMLADHX.S32  Q3, Q0, Q1  // Imaginary parts of FXMULTR(Q0, Q1),
                            // real parts are untouched
                            // Q3[2*i+1] = Q0[2*i+1]*Q1[2*i] + Q0[2*i]*Q1[2*i+1]  i={0..1}
                            // Q3 = [ 0 33554433 0 33554433 ]

// FXMULTR=saturated multiplication with doubling and rounding high part extraction
```

3.29.45 VQDMLAH, VQRDMLAH (vector by scalar plus vector)

Vector Saturating Doubling Multiply Accumulate, Vector Saturating Rounding Doubling Multiply Accumulate.

Syntax

```
VQDMLAH<v>.<dt> Qda, Qn, Rm
VQRDMLAH<v>.<dt> Qda, Qn, Rm
```

Parameters

Qda Accumulator vector register.

Qn Source vector register.

Rm Source general-purpose register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values

- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

Updates QC flag in FPSCR register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Multiply each element in the source vector by a scalar value, double the result and add to the respective element from the higher half of the destination vector. Store the higher half of each result in the destination register. The result is optionally rounded before the higher half is selected and saturated.

VQDMLAH (vector by scalar plus vector) example

```
// 16-bit integer Vector Saturating Doubling Multiply Accumulate (vector by scalar plus
vector)
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [R1]     // 16-bit vector contiguous load
                             // Q0 = [  4096 8192  12288 16384  20480
                             // 24576  28672 -32768  ]
                             // Q1 = [  -4096 12288  -20480 28672
                             // 28672 -20480  12288 -4096  ]
MOV             R0, #0x4000  // 0.5 in Q.15

VQDMLAH.S16     Q0, Q1, R0   // Q0[i] = SSAT16(Q0[i] + FXMULT(Q1[i], R0))  i={0..7}
                             // Q1 = [  2048 14336  2048 30720  32767
                             // 14336  32767 -32768  ]
```

VQRDMLAH (vector by scalar plus vector) example

```
// 16-bit integer Vector Rounding Saturating Doubling Multiply Accumulate (vector by scalar
plus vector)
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [R1]     // 16-bit vector contiguous load
                             // Q0 = [  4096 8192  12288 16384
                             // 20480 24576  28672 -32768  ]
                             // Q1 = [  -4096 12288  -20480 28672
                             // 28672 -20480  12288 -4096  ]
MOV             R0, #0x4000  // 0.5 in Q.15

VQRDMLAH.S16    Q0, Q1, R0   // Q0[i] = SSAT16(Q0[i] + FXMULTR(Q1[i], R0))   i={0..7}
                             // Q0 = [  2048 14336  2048 30720
                             // 32767 14336  32767 -32768  ]

// FXMULTR=saturated multiplication with doubling and rounding high part extraction
```

3.29.46 VQDMLASH, VQRDMLASH (vector by vector plus scalar)

Vector Saturating Doubling Multiply Accumulate Scalar High Half, Vector Saturating Rounding Doubling Multiply Accumulate Scalar High Half.

Syntax

```
VQDMLASH<v>.<dt> Qda, Qn, Rm
VQRDMLASH<v>.<dt> Qda, Qn, Rm
```

Parameters

Qda Source and destination vector register.

Qn Source vector register.

Rm Source general-purpose register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:
- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

Updates QC flag in FPSCR register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Multiply each element in the source vector by the respective element from the destination vector, double the result and add to a scalar value. Store the higher half of each result in the destination register. The result is optionally rounded before the higher half is selected and saturated.

VQDMLASH (vector by vector plus scalar) example

```
// 16-bit integer Vector Saturating Doubling Multiply Accumulate Scalar High Half (vector by
vector plus scalar)
VLDRH.16        Q0, [r0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [r1]     // 16-bit vector contiguous load
                             // Q0 = [  4097 8193  12289 16385  20481
                             // 24577  28673 -32767  ]
                             // Q1 = [  -4096 12288  -20480 28672  28672
                             // -20480  12288 -4096  ]
MOV             R0, #0x4000  //  Accumulator = 0.5 in Q.15

VQDMLASH.S16    Q0, Q1, R0   // Q0[i] = SSAT16(R0 + FXMULT(Q0[i], Q1[i]))  i={0..7}
                             // Q0 (output) = [ 15871 19456  8703 30720
                             // 32767 1023  27136 20479  ]
```

VQRDMLASH (vector by vector plus scalar) example

```
// 16-bit integer Vector SaturatingRounding  Doubling Multiply Accumulate Scalar High Half
(vector by vector plus scalar)
VLDRH.16         Q0, [R0]      // 16-bit vector contiguous load
VLDRH.16         Q1, [R1]      // 16-bit vector contiguous load
                               // Q0 = [  4097 8193  12289 16385
                               // 20481 24577  28673 -32767  ]
                               // Q1 = [  -4095 12289  -20479 28673
                               // 28673 -20479  12289 -4095  ]
MOV              R0, #0x4000   //  Accumulator = 0.5 in Q.15

VQRDMLASH.S16    Q0, Q1, R0    // Q0[i] = (SSAT(R0 + 2 * (Q0[i] * Q1[i])
                               // + (1 << 15))) >> 16  i = {0..7}
                               // Q0 (output) = [  15872 19457 8704 30721
                               // 32767 1024 27137 20479   ]
```

3.29.47 VQDMLSDH, VQRDMLSDH

Vector Saturating Doubling Multiply Subtract Dual Returning High Half, Vector Saturating Rounding Doubling Multiply Subtract Dual Returning High Half.

Syntax

```
VQRDMLSDH{X}<v>.<dt> Qd, Qn, Qm
VQDMLSDH{X}<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:
- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

Updates QC flag in FPSCR register.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other and doubling the result. The higher halves of the resulting values are selected as the final results. The base variant writes the results into the lower element of each pair of elements in the destination register, whereas the exchange variant writes to the upper element in each pair. The results are optionally rounded before the higher half is selected and saturated.

VQDMLSDH example

```
// Vector Saturating Doubling Multiply Subtract Dual Returning High Half
VLDRW.32        Q0, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q1, [R1]    // 32-bit contiguous vector load
                            // Q0 = [ 268435456 536870912 805306368 1073741824 ]
                            // Q1 = [ -268435456 805306368 -1342177280 1879048192 ]
VMOV.S32        Q2, #0

VQDMLSDH.S32    Q2, Q0, Q1  // real parts of FXMULT(Q0, Q1)),
                            // imaginary parts are untouched
                            // Q2[2*i] = Q0[2*i]*Q1[2*i] - Q0[2*i+1]*Q1[2*i+1]
                            // i={0..1}
                            // Q2 =[ -234881024 0 -1442840576 0 ]

// Exchange version
VMOV.S32        Q3, #0
                            // Q0 = [ 268435456 536870912 805306368 1073741824 ]
                            // Q1 = [ -268435456 805306368 -1342177280 1879048192 ]
VQDMLSDHX.S32   Q3, Q0, Q1  // Imaginary parts of FXMULT(Q0, conjugate(Q1)),
                            // real parts are untouched
                            // Q3[2*i+1] = Q0[2*i+1]*Q1[2*i] - Q0[2*i]*Q1[2*i+1]
                            // i={0..1}
                            // Q3 = [ 0 -167772160 0 -1375731712 ]

// FXMULT=saturated multiplication with doubling and high part extraction
```

VQRDMLSDH example

```
// 32-bit integer Vector Rounding Saturating Doubling Multiply Subtract Dual Returning High
Half
VLDRW.32        Q0, [R0]    // 32-bit contiguous vector load
VLDRW.32        Q1, [R1]    // 32-bit contiguous vector load
                            // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                            // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]
VMOV.S32        Q2, #0
VQRDMLSDH.S32   Q2, Q0, Q1  // Real parts of FXMULTR(Q0, Q1)),
                            // imaginary parts are untouched
                            // Q2[2*i] = Q0[2*i]*Q1[2*i] - Q0[2*i+1]*Q1[2*i+1]
                            // Q2 =[ -234881025 0 -1442840578 0 ]

// Exchange variant
VMOV.S32        Q3, #0
                            // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                            // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]
VQRDMLSDHX.S32  Q3, Q0, Q1  // Imaginary parts of FXMULTR(Q0, conj(Q1)),
                            // real parts are untouched
                            // Q3[2*i+1] = Q0[2*i+1]*Q1[2*i] - Q0[2*i]*Q1[2*i+1]
                            // Q3 = [ 0 -167772160 0 -1375731713 ]

// FXMULTR=saturated multiplication with doubling, rounding and high part extraction
```

3.29.48 VQDMULH, VQRDMULH

Vector Saturating Doubling Multiply Returning High Half, Vector Saturating Rounding Doubling Multiply Returning High Half.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VQDMULH<v>.<dt> Qd, Qn, Rm
VQRDMULH<v>.<dt> Qd, Qn, Rm
VQRDMULH<v>.<dt> Qd, Qn, Qm
VQDMULH<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register. Source vector register.

Rm Source general-purpose register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

Updates QC flag in FPSCR register.

Operation

Multiply a general-purpose register value by each element of a vector register to produce a vector of results or multiply each element of a vector register by its corresponding element in another vector register, double the results, and place the most significant half of the final results in the destination vector. The results are optionally rounded before being saturated.

VQDMULH example

```
// 16-bit integer Vector Saturating Doubling Multiply Returning High Half
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [R1]     // 16-bit vector contiguous load
                             // Q0 = [ 32767 8193  12289 16385
                             // 20481 24577  28673 -32768  ]
                             // Q1 = [ 32767 12288  -20480 28672
                             // 28672 -20480  12288 -32768  ]
MOV             R0, #0x4000  // 0.5 in Q.15
// Vector by scalar variant
VQDMULH.S16     Q2, Q0, R0   // Q2[i] = SSAT(2 * Q0[i] * R0) >> 16    i={0..7}
                             // Q2 = [  16383 4096  6144 8192
                             // 10240 12288  14336 -16384  ]
// Vector by vector variant
VQDMULH.S16     Q3, Q0, Q1   // Q3[i] = SSAT(2 * Q0[i] * Q1[i]) >> 16  i={0..7}
                             // Q3 = [  32766 3072  -7681 14336
                             // 17920 -15361  10752 32767  ]

// 16-bit integer Vector Saturating Doubling Multiply Returning High Half (rounding variants)
MOV             R0, #0x4000   // 0.5 in Q.15
                              // Q0 = [ 32767 8193  12289 16385
                              // 20481 24577  28673 -32768  ]
                              // Q1 = [ 32767 12288  -20480 28672
                              // 28672 -20480  12288 -32768  ]
// Vector by scalar variant
VQRDMULH.S16    Q2, Q0, R0    // Q2[i] = SSAT(2 * Q0[i] * R0
                              // + (1<<15)) >> 16   i={0..7}
                              // Q2 = [  16384 4097  6145 8193
                              // 10241 12289  14337 -16384  ]
// Vector by vector variant
VQRDMULH.S16    Q3, Q0, Q1    // Q3[i] = SSAT(2 * Q0[i] * Q1[i]
                              //  + (1<<15)) >> 16  i={0..7}
                              // Q3 = [  32766 3072  -7681 14337
                              // 17921 -15361  10752 32767  ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VQRDMULH example

```
//  16-bit integer Vector Saturating Rounding Doubling Multiply Subtract Dual Returning High
Half
VLDRH.16         Q0, [R0]      // 16-bit vector contiguous load
VLDRH.16         Q1, [R1]      // 16-bit vector contiguous load
                               // Q0 = [  32767 8193  12289 16385  20481 24577  28673
-32768  ]
                               // Q1 = [  32767 12288  -20480 28672  28672 -20480  12288
-32768  ]
MOV              R0, #0x4000   // 0.5 in Q.15
// Vector by vector variant
VQDMULH.S16      Q2, Q0, Q1    // Q2[i] = FXMULT(Q0[i], Q1[i])   i={0..7}
                               // Q2 = [  32766 3072  -7681 14336  17920 -15361  10752
32767  ]
// Vector by scalar variant
VQDMULH.S16      Q3, Q0, R0    // Q3[i] = FXMULT(Q0[i], R0)      i={0..7}
                               // Q3 = [  16383 4096  6144 8192  10240 12288  14336 -16384  ]

// Rounding variants

                              // Q0 = [  32767 8193  12289 16385  20481 24577  28673
-32768  ]
                              // Q1 = [  32767 12288  -20480 28672  28672 -20480  12288
-32768  ]
MOV           R0, #0x4000
// Vector by vector variant
VQRDMULH.S16  Q2,Q0,Q1        // Q2[i] = FXMULTR(Q0[i], Q1[i])
                              // Q2 = [  32766 3072  -7681 14337  17921 -15361  10752
32767  ]
// Vector by scalar variant
VQRDMULH.S16  Q3,Q0,R0        // Q2[i] = FXMULTR(Q0[i], R0)
                              // Q3 = [  16384 4097  6145 8193  10241 12289  14337 -16384  ]
```

3.29.49 VQDMULL

Vector Multiply Long.

Syntax

```
VQDMULL<T><v>.<dt> Qd, Qn, Qm
VQDMULL<T><v>.<dt> Qd, Qn, Rm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

Rm Source general-purpose register.

T Specifies which half of the source element is used. This parameter must be one of the following values:
- B, indicates bottom half.
- T, indicates top half.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

- When S32 is used, then Qd must not use the same register as Qm
- Rm must not use the same register as SP and PC
- When S32 is used, then Qd must not use the same register as Qn

Post-conditions

Updates QC flag in FPSCR register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Performs an element-wise integer multiplication of two single-width source operand elements. These are selected from either the top half (T variant) or bottom half (B variant) of double-width source vector register elements or the lower single-width portion of the general-purpose register. The product of the multiplication is doubled and saturated to produce a double-width product that is written back to the destination vector register.

VQDMULL example

```
// 16-bit integer vector saturared, doubled multiply long
VLDRH.16        Q0, [R0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [R1]     // 16-bit vector contiguous load
                             // Q0 = [  32767 8193  12289 16385
                             // 20481 24577  28673 -32768  ]
                             // Q1 = [  32767 12288  -20480 28672
                             // 28672 -20480  12288 -32768  ]

// Multiply bottom parts (of adjacent pairs source)
VQDMULLB.S16    Q2, Q0, Q1   // Q2[i] = SSAT(2 * Q0[2*i] * Q1[2*i])   i={0..3}
                             // Q2 = [ 2147352578 -503357440 1174462464 704667648 ]

// Multiply top parts (of adjacent pairs source)
VQDMULLT.S16    Q3, Q0, Q1   // Q3[i] = SSAT(2 * Q0[2*i+1] * Q1[2*i+1])   i={0..3}
                             // Q3 = [ 201351168 939581440 -1006673920 2147483647 ]
```

3.29.50 VQNEG

Vector Saturating Negate.

Syntax

```
VQNEG<v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- S8
- S16
- S32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Updates QC flag in FPSCR register.

Operation

Negate the value and saturate each element in a vector register.

VQNEG example

```
// Vector Saturating Negate.
VLDRH.16             Q0, [R0]   // 16-bit contiguous load
                                // Q0 = [  32767 12288  -20480 28672
                                // 28672 -20480  12288 -32768  ]

VQNEG.S16            Q1, Q0     // Q1[i] = SSAT(-Q0[i])   i={0..7}
                                // Q1 = [  -32767 -12288  20480 -28672
                                // -28672 20480  -12288 32767  ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.29.51 VQSUB

Vector Saturating Subtract.

Syntax

```
VQSUB<v>.<dt> Qd, Qn, Rm
VQSUB<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt This parameter determines the following values:
- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

Rm must not use the same register as SP and PC

Post-conditions

Updates QC flag in FPSCR register.

Operation

Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register. The result is saturated before being written to the destination vector register.

VQSUB example

```
// 16-bit integer Vector Saturating Subtract.
VLDRH.16    Q0, [R0]    // 16-bit contiguous load
VLDRH.16    Q1, [R1]    // 16-bit vector contiguous load
                        // Q0 = [  32767 12288  -20480 28672
                        // 28672 -20480  12288 -32768  ]
                        // Q1 = [  32767 8193  12289 16385
                        // 20481 24577  28673 -32768  ]
MOV         R0,  0x4000  // 0.5 in Q.15

// Vector by register
VQSUB.S16   Q2, Q0, R0  // Q2[i] = SSAT(Q0[i] - Q0)  i=(0..7}
                        // Q2 = [  16383 -4096  -32768 12288
                        // 12288 -32768  -4096 -32768  ]

// Vector by vector
VQSUB.S16  Q2, Q0, Q1    // Q2[i] = SSAT(Q0[i] - Q1[i])   i=(0..7}
                         // Q2 = [  0 4095  -32768 12287  8191
                         // -32768  -16385 0  ]
```

3.29.52 VRHADD

Vector Rounding Halving Add.

Syntax

```
VRHADD<v>.<dt> Qd, Qn, Qm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Add the value of the elements in the first source vector register to the respective elements in the second source vector register. The result is halved and rounded before being written to the destination vector register.

VRHADD example

```
// 16-bit integer Vector Rounding Halving Add.
VLDRH.16        Q0, [r0]     // 16-bit vector contiguous load
VLDRH.16        Q1, [R1]     // 16-bit vector contiguous load
                             // Q0 = [  -32768 12288  -20480 28672
                             // 28672 -20480  12288 32767  ]
                             // Q1 = [  4097 32767  12289 16385
                             // 20481 24577  28673 -32768  ]

VRHADD.S16      Q2, Q0, Q1   // Q2[i] = (Q0[i] + Q1[i] + 1) >> 1    i={0..7}
                             // Q2 = [  -14335 22528  -4095 22529
                             // 24577 2049  20481 0  ]
```

3.29.53 VRINT (floating-point)

Vector Round Integer.

Syntax

```
VRINT<op><v>.<dt> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

dt Indicates the floating-point format used. This parameter must be one of the following values

- F16
- f32

op The rounding mode. This parameter must be one of the following values:

- N round to nearest with ties to even X round to nearest with ties to even.
- Raising inexact exception if result not numerically equal to input A round to nearest with ties to away Z round towards zero M round towards minus infinity P round towards plus infinity.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

v Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Round a floating-point value to an integer value. The result remains in floating-point format. It is not converted to an integer.

VRINT (floating-point) example

```
// Vector Round Integer (remains float)
MOV           R0, #1
MOV           R1, #0x33AE  // 0.24f16
VIDUP.U16     Q0, R0, #1   // Generates 16-bit incrementing sequence, starting at 1 with
increments of 1
VCVT.F16.S16  Q0, Q0       // Convert S16 vector into F16 vector
VMUL.F16      Q0, Q0, 1   // Q0[i] = Q0[i] * 0.24f16    i={0..7}

// Z, Round towards zero
                           // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                           // 1.200195 1.439453  1.679688 1.919922 ]
VRINTZ.F16    Q1,Q0        // Q1[i] = RND(Q0[i], Z)    i={0..7}
                           // Q1 = [ 0.0 0.0  0.0 0.0  1.0 1.0  1.0 1.0 ]

// A, Round to nearest, with ties away
                           // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                           // 1.200195 1.439453  1.679688 1.919922 ]
VRINTA.F16    Q1,Q0        // Q1[i] = RND(Q0[i], A)    i={0..7}
                           // Q1 = [ 0.0 0.0  1.0 1.0  1.0 1.0  2.0 2.0 ]

//X, Round to nearest with ties to even, raising inexact exception if result not numerically
equal to input
                           // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                           // 1.200195 1.439453  1.679688 1.919922 ]
VRINTX.F16    Q1,Q0        // Q1[i] = RND(Q0[i], X)    i={0..7}
                           // Q1 = [ 0.0 0.0  1.0 1.0  1.0 1.0  2.0 2.0 ]

// N, Round to nearest, with ties to even
                          // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                          // 1.200195 1.439453  1.679688 1.919922 ]
VRINTN.F16   Q1,Q0        // Q1[i] = RND(Q0[i], N)    i={0..7}
                          // Q1 = [ 0.0 0.0  1.0 1.0  1.0 1.0  2.0 2.0 ]

// M, Round towards Minus Infinity
                          // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                           // 1.200195 1.439453  1.679688 1.919922 ]
VRINTM.F16   Q1,Q0        // Q1[i] = RND(Q0[i], M)    i={0..7}
                          // Q1 = [ 0.0 0.0  0.0 0.0  1.0 1.0  1.0 1.0 ]

// P, Round towards Plus Infinity
                          // Q0 = [ 0.239990 0.479980  0.719727 0.959961
                          // 1.200195 1.439453  1.679688 1.919922 ]
VRINTP.F16   Q1,Q0        // Q1[i] = RND(Q0[i], P)    i={0..7}
                          // Q1 = [ 1.0 1.0  1.0 1.0  2.0 2.0  2.0 2.0 ]
```

3.29.54 VRMLALDAVH

Vector Rounding Multiply Add Long Dual Accumulate Across Vector Returning High 64 bits.

Syntax

```
VRMLALDAVH{A}{X}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- `A`.

Qm Second source vector register.

Qn First source vector register.

RdaHi General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

RdaLo General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- `X`.

dt This parameter must be one of the following values:
- `S32`
- `U32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by adding them together. At the end of each beat these results are accumulated. The upper 64 bits of a 72-bit accumulator value is selected and stored across two registers, the top 32 bits are stored in an even-numbered register and the lower 32 bits are stored in an odd-numbered register. The initial value of the general-purpose destination registers can optionally be shifted up by 8 bits and added to the result. The result is rounded before the top 64 bits are selected.

VRMLALDAVH example

```
// Vector Rounding Multiply Add Long Dual Accumulate Across Vector Returning High 64 bits.
VLDRW.s32       Q0, [R0]        // 32-bit contiguous vector load. r0 points to buffer
containing
                                // {0.125, 0.25, 0.375, 0.5} in Q.31
VLDRW.S32       Q1, [R0, #16]   // 32-bit contiguous vector load.
                                // r0 points to buffer containing
                                // {-0.125, 0.375, -0.625 0.875} in Q.31
                                // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                                // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]

VRMLALDAVH.S32  R0,R1,Q0,Q1     // R0:R1 = (sum(Q0[i] * Q1[i]) + (1<<7)) >> 8
                                // i={0..3}
                                // R0:R1 (out) 5066549595471872 (=0.28125 in Q.54)

// Accumulated variant
VRMLALDAVHA.S32   R0,R1,Q0,Q1   // R0:R1 += (sum(Q0[i] * Q1[i]) + (1<<7)) >> 8
                                // i={0..3}
                                // R0:R1 (in) 5066549595471872  (from previous vrmlaldavh)
                                // R0:R1 (out) 10133099190943744 (= 0.5625 in Q.54)

// Exchange variant
VRMLALDAVHAX.S32   R0,R1,Q0,Q1   // R0:R1 = (sum(Q0[2*i] * Q1[2*i+1] + Q0[2*i+1] * Q1[2*i])
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
+ (1<<7)) >> 8    i={0,1}
                               // R0:R1 (out) 562949968101376 (=0.03125 in Q.54)
```

3.29.55 VRMLALVH

Vector Multiply Accumulate Long Across Vector Returning High 64 bits.

Syntax

```
VRMLALVH{A}<v>.<dt> RdaLo, RdaHi, Qn, Qm
```

Parameters

None.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

This is an alias of VRMLALDAVH without exchange.

This is an alias of VRMLALDAVH with the following condition satisfied: `X == 0`.

VRMLALVH example

```
// Vector Multiply Accumulate Long Across Vector Returning High 64 bits (= alias of
VRMLALDAVH without exchange)
VLDRW.S32       Q0, [R0]        // 32-bit contiguous vector load.
                                // r0 points to buffer containing
                                // {0.125, 0.25, 0.375, 0.5} in Q.31
VLDRW.S32       Q1, [R0, #16]   // 32-bit contiguous vector load.
                                // r0 points to buffer containing
                                // {-0.125, 0.375, -0.625 0.875} in Q.31
                                // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                                // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]

VRMLALVH.S32    R0,R1,Q0,Q1     // R0:R1 = (sum(Q0[i] * Q1[i]) + (1<<7)) >> 8    i={0..3}
                                // R0:R1 (out) 5066549595471872 (=0.28125 in Q.54)

// Accumulated variant
VRMLALVHA.S32     R0,R1,Q0,Q1   // R0:R1 += (sum(Q0[i] * Q1[i]) + (1<<7)) >> 8    i={0..3}
                                // R0:R1 (in) 5066549595471872 (from previous VRMLALVH)
                                // R0:R1 (out) 10133099190943744 (= 0.5625 in Q.54)
```

3.29.56 VRMLSLDAVH

Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bits.

Syntax

```
VRMLSLDAVH{A}{X}<v>.S32 RdaLo, RdaHi, Qn, Qm
```

Parameters

A Accumulate with existing register contents. This parameter must be one of the following values:
- Parameter unlisted.
- A.

Qm Second source vector register.

Qn First source vector register.

RdaHi General-purpose register for the high-half of the 64-bit source and destination. This must be an odd numbered register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

RdaLo General-purpose register for the low-half of the 64 bit source and destination. This must be an even numbered register.

X Exchange adjacent pairs of values in Qm. This parameter must be one of the following values:
- Parameter unlisted.
- X.

v See Standard Assembler Syntax Fields

Restrictions

RdaHi must not use SP

Post-conditions

There are no condition flags.

Operation

The elements of the vector registers are handled in pairs. In the base variant, corresponding elements from the two source registers are multiplied together, whereas the exchange variant swaps the values in each pair of values read from the first source register, before multiplying them with the values from the second source register. The results of the pairs of multiply operations are combined by subtracting one from the other. At the end of each beat these results are accumulated. The upper 64 bits of a 72-bit accumulator value is selected and stored across two registers, the top 32 bits are stored in an even-numbered register and the lower 32 bits are stored in an odd-numbered register. The initial value of the general-purpose destination registers can optionally be shifted up by 8 bits and added to the result. The result is rounded before the top 64 bits are selected.

VRMLSLDAVH example

```
// Vector Rounding Multiply Subtract Long Dual Accumulate Across Vector Returning High 64 bit
VLDRW.s32       Q0, [R0]        // 32-bit contiguous vector load.
                                // r0 points to buffer containing
                                // {0.125, 0.25, 0.375, 0.5} in Q.31
VLDRW.S32       Q1, [R0, #16]   // 32-bit contiguous vector load.
                                // r0 points to buffer containing
                                // {-0.125, 0.375, -0.625 0.875} in Q.31
                                // Q0 = [ 268435457 536870913 805306369 1073741825 ]
                                // Q1 = [ -268435455 805306369 -1342177279 1879048193 ]

VRMLSLDAVH.S32  R0,R1,Q0,Q1     // R0:R1 = (sum(Q0[2*i] * Q1[2*i]
                                // - Q0[2*i+1] * Q1[2*i+1]) + (1<<7)) >> 8
                                //    i = {0, 1}
                                // R0:R1 (out) -14073748854407168 (= -0.78125 in Q.54)
                                // = (0.125 * -0.125) - (0.25 * 0.375)
                                // + (0.375 * -0.625) - (0.5 * 0.875) (Q.31 data)
                                // = -0.015625 - 0.09375 -0.234375 - 0.4375
                                // = -0.78125
// Accumulated variant
VRMLSLDAVHA.S32   R0,R1,Q0,Q1   // R0:R1 = R0:R1 - (sum(Q0[2*i]
                                // * Q1[2*i] - Q0[2*i+1] * Q1[2*i+1]) + (1<<7)) >> 8
                                //  i = {0, 1}
                                // R0:R1 (in) -14073748854407168
                                //  (= -0.78125 in Q.54, from previous VRMLSLDAVH)
                                // R0:R1 (out) -28147497708814336 (= -1.5625 in Q.54)

// exchange variant
VRMLSLDAVHX.S32   R0,R1,Q0,Q1   // R0:R1 = (sum(Q0[2*i+1] * Q1[2*i]
                                 // - Q0[2*i] * Q1[2*i+1]) + (1<<7)) >> 8  i = {0, 1}
                                // R0:R1 (out) -12947848943370240 (= -0.71875 in Q.54)
```

3.29.57 VSBC

Whole Vector Subtract With Carry.

Syntax

```
VSBC{I}<v>.I32 Qd, Qn, Qm
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

I Specifies where the initial carry in for wide arithmetic comes from. This parameter must be one of the following values:
- Parameter unlisted, indicates carry input comes from FPSCR.C.
- `I`, indicates carry input is 1.

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

Reads C flag in FPSCR and updates the N, Z, C, and V flags in FPSCR register.

Operation

Beat-wise subtracts the value of the elements in the second source vector register and the value of ! FPSCR.C from the respective elements in the first source vector register, the carry flag being FPSCR.C. The initial value of FPSCR.C can be overridden by using the I variant. FPSCR.C is not updated for beats disabled due to predication. FPSCR.N, FPSCR.V and FPSCR.Z are zeroed.

VSBC example

```
// Whole Vector Subtract With Carry
MOV         R0, #2
MOV         R1, #1
VMOV.U32    Q0, #0       // Clear Q0
VMOV.U32    Q1, #0       // Clear Q1
VMOV.U32    Q0[3], R0    // Q[3] = 2
VMOV.U32    Q1[2], R1    // Q[2] = 1
                         // Q0 =[ 00000000 00000000 00000000 00000002 ]
                         // = 0x00000002000000000000000000000000

                         // Q1 =[ 00000000 00000000 00000001 00000000 ]
                         // = 0x00000000000000010000000000000000


VSBCI.I32   Q2, Q0, Q1   // Q2 = Q0 - Q1, carry input is 1.
                         // Q2 =[ 00000000 00000000 FFFFFFFF 00000001 ]
                         // = 0x00000001FFFFFFFF0000000000000000
```

3.29.58 VSUB

Vector Subtract.

Syntax

```
VSUB<v>.<dt> Qd, Qn, Rm
VSUB<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- `I8`
- `I16`
- `I32`

v See Standard Assembler Syntax Fields

Restrictions

`Rm` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Subtract the value of the elements in the second source vector register from the respective elements in the first source vector register. The result is then written to the destination vector register.

VSUB example

```
// 16-bit integer Vector Subtract
MOV           R0, #1
MOV           R1, #1000
VIDUP.U16     Q0, R0, #1    // generates incrementing sequence,
                            //  starting at 1 with increments of 1
VMUL.S16      Q0, Q0, R1    // multiply by 1000
MOVW          R0, #4000
                            // Q0 = [  1000 2000  3000 4000  5000 6000  7000 8000  ]
// Vector by scalar variant
VSUB.I16      Q1, Q0, R0    // Q1[i] = Q0[i] - R0 i={0..7}
                            // Q1 = [  -3000 -2000  -1000 0  1000 2000  3000
4000  ]

MOV           R0, #0
VIDUP.U16     Q1, R0, #2    // Generates incrementing sequence,
                            // starting at 1 with increments of 1
VMUL.S16      Q1, q1, R1    // Multiply by 1000
                            // Q0 = [  1000 2000  3000 4000  5000 6000  7000 8000  ]
                            // Q1 = [  0 2000  4000 6000  8000 10000  12000 14000  ]

// Vector by vector variant
VSUB.I16      Q2, Q0, Q1    // Q2[i] = Q0[i] - Q1[i] i={0..7}
                            // Q2 = [  1000 0  -1000 -2000  -3000 -4000  -5000 -6000  ]
```

3.29.59 VSUB (floating-point)

Vector Subtract.

Syntax

```
VSUB<v>.<dt> Qd, Qn, Rm
VSUB<v>.<dt> Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Second source vector register.

Qn First source vector register.

Rm Source general-purpose register.

dt Indicates the floating-point format used. This parameter must be one of the following values:

- `F32`
- `F16`

v See Standard Assembler Syntax Fields

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

`Rm` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Subtract the value of the elements in the second source vector register from either the respective elements in the first source vector register or a general-purpose register.

VSUB (floating-point) example

```
// 16-bit float Vector Subtract Float.
MOV           R0, #1
MOV           R1, #1000
VIDUP.U16     Q0, R0, #1    // Generates incrementing sequence,
                            // starting at 1 with increments of 1
VMUL.S16      Q0, Q0, R1    // Multiply by 1000
VCVT.F16.S16  Q0, Q0        // Convert into F16 vector
MOVW          R0, #0x6BD0   // 4000.0f16
                            // Q0 = [  1000.0  2000.0   3000.0 4000.0
                            //  5000.0 6000.0  7000.0 8000.0 ]

// Vector by scalar variant
VSUB.F16      Q1, Q0, R0    // Q1[i] = Q0[i] - R0 i={0..7}
                            // Q1 = [ -3000.0 -2000.0  -1000.0
                            //    0.0  1000.0 2000.0  3000.0 4000.0 ]

MOV           R0, #0
VIDUP.U16     Q1, R0, #2    // Generates incrementing sequence,
                            // starting at 1 with increments of 1
VMUL.S16      Q1, Q1, R1    // Multiply by 1000
VCVT.F16.S16  Q1, Q1        // Convert into F16 vector

                            // Q0 = [ 1000.0 2000.0  3000.0
                            //  4000.0  5000.0  6000.0   7000.0  8000.0 ]
                            // Q1 = [    0.0 2000.0  4000.0  6000.0
                            //  8000.0 10000.0  12000.0 14000.0 ]
// Vector by vector variant
VSUB.F16      Q2, Q0, Q1    // Q2[i] = Q0[i] - Q1[i] i={0..7}
                            // Q2 = [ 1000.0    0.0 -1000.0 -2000.0
                            // -3000.0 -4000.0  -5000.0 -6000.0 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.30 Arm®v8.1-M vector bitwise operations instructions

Reference material for the Cortex-M55 processor vector bitwise operations instructions.

Note

This document 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. 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 your relevant assembler documentation for these details.

3.30.1 List of Arm®v8.1-M vector bitwise operations instructions

An alphabetically ordered list of the Armv8.1-M vector bitwise operations instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-23 Armv8.1-M bitwise operations instructions

Mnemonic	Brief description	See
VAND	Vector Bitwise And	*3.30.2 VAND* on page 3-409
VAND	VAND (immediate) Vector Bitwise And	*3.30.3 VAND (immediate)* on page 3-410
VBIC	VBIC (immediate) Vector Bitwise Clear	*3.30.4 VBIC (immediate)* on page 3-411
VBIC	VBIC (register) Vector Bitwise Clear	*3.30.5 VBIC (register)* on page 3-411
VEOR	Vector Bitwise Exclusive Or	*3.30.6 VEOR* on page 3-412
VMOV	VMOV (immediate) Vector Move	*3.30.7 VMOV (immediate)* on page 3-413
VMOV	VMOV (register) Vector Move	*3.30.8 VMOV (register)* on page 3-414
VMOV	VMOV Vector Move (general-purpose register to vector lane)	*3.30.9 VMOV (general-purpose register to vector lane)* on page 3-415
VMOV	VMOV Vector Move (vector lane to general-purpose register)	*3.30.10 VMOV (vector lane to general-purpose register)* on page 3-415
VMVN	VMVN (immediate) Vector Bitwise NOT	*3.30.11 VMVN (immediate)* on page 3-416
VMVN	VMVN (register) Vector Bitwise NOT	*3.30.12 VMVN (register)* on page 3-417
VORN	Vector Bitwise Or Not	*3.30.13 VORN* on page 3-417
VORN	VORN (immediate) Vector Bitwise Or Not	*3.30.14 VORN (immediate)* on page 3-418
VORR	Vector Bitwise Or	*3.30.15 VORR* on page 3-419
VORR	VORR (immediate)Vector Bitwise Or	*3.30.16 VORR (immediate)* on page 3-420
VPSEL	Vector Predicated Select	*3.30.17 VPSEL* on page 3-420
VREV16	Vector Reverse	*3.30.18 VREV16* on page 3-421
VREV32	Vector Reverse	*3.30.19 VREV32* on page 3-422
VREV64	Vector Reverse	*3.30.20 VREV64* on page 3-423

3.30.2 VAND

Vector Bitwise And.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VAND<v>{.<dt>} Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

dt An optional data type. It is ignored by assemblers and does not affect the encoding. This parameter can be one of the following:

- S8
- S16
- S32
- U8
- U16
- U32
- I8
- I16
- I32
- F16
- F32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise AND of a vector register with another vector register. The result is written to the destination vector register.

VAND example

```
// Compute a bitwise AND of 2 x 8-bit vector registers
MOV            R0, #0
VMOV.I8        Q1, #0x0F
VIDUP.U8       Q0, R0, #4   // Generates an 8-bit incrementing sequence,
                            // starting at 0 with increments of 4
                            // Q0 = [ 0x00 0x04 0x08 0x0C 0x10 0x14 0x18
                            // 0x1C 0x20 0x24 0x28 0x2C 0x30 0x34 0x38 3C] (int8x16_t)
                            // Q1 = [ 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F
                            // 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F 0x0F]
VAND           Q2, Q0, Q1   // Q2 = Q0 & Q1; (data type can be eluded)
                            // Q2 = [ 0x00 0x04 0x08 0x0C 0x00 0x04 0x08 0x0C
                            // 0x00 0x04 0x08 0x0C 0x00 0x04 0x08 0x0C]
```

3.30.3 VAND (immediate)

Vector Bitwise AND.

Syntax

```
VAND<v>.<dt> Qda, #<imm>
is equivalent to
VBIC<v>.<dt> Qda, #~<imm>
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

None.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

This is a pseudo-instruction, equivalent to a VBIC (immediate) instruction with the immediate value bitwise inverted, and VAND (immediate) is never the preferred disassembly. See *3.30.4 VBIC (immediate)* on page 3-411 for an example.

3.30.4 VBIC (immediate)

Vector Bitwise Clear.

Syntax

```
VBIC<v>.<dt> Qda, #<imm>
```

Parameters

Qda Source and destination vector register.

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - I32
  - I16

imm The immediate value to load in to each element.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise AND of a vector register and the complement of an immediate value.

VBIC (immediate) example

```
// Clear the 2 LSBs of a 16-bit integer vector register
MOV           R2, #100
VIDUP.U16     Q0, R2, #1    // Generates incrementing sequence,
                            // starting at 100 with increments of 1
                            // Q0 (in)  = [ 0x0064 0x0065 0x0066 0x0067
                            // 0x0068 0x0069 0x006A 0x006B ]
VBIC.I16      Q0, #3        // Q0[i] = Q0[i] & (~3)  i={0..7},
                            // clear last 2 bits of the 16-bit elements vector
                            // Q0 (out) = [ 0x0064 0x0064 0x0064 0x0064
                            // 0x0068 0x0068 0x0068 0x0068 ]
```

3.30.5 VBIC (register)

Vector Bitwise Clear.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Syntax

```
VBIC<v>{.<dt>} Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

dt An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following:

- S8
- S16
- S32
- U8
- U16
- U32
- I8
- I16
- I32
- F16
- F32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise AND of a vector register and the complement of a vector register.

VBIC (register) example

```
// Clear the LSBs of a 16-bit integer vector register based on an incrementing mask vector
MOV          R0, #0
MOV          R1, #0xFFFF
MOV          R2, #1
VDUP.16      Q0, R1       // Duplicate 0xFFFF over 16-bit vector
VIDUP.U16    Q1, R0, #1   // Incrementing 16-bit vector starting at 0 with increments of 1
VDUP.16      Q2, R2       // Duplicate 0x1 over 16-bit vector
VSHL.S16     Q1, Q2, Q1   // Q1[i] = Q2[i] << Q1[i]     i={0..7}
VSUB.S16     Q1, Q1, R2   // Q1[i] = Q1[i] - 1          i={0..7}
                          // Q0 = [ 0xFFFF 0xFFFF 0xFFFF 0xFFFF 0xFFFF 0xFFFF
                          // 0xFFFF 0xFFFF ]
                          // Q1 = [ 0x0000 0x0001 0x0003 0x0007 0x000F 0x001F
                          // 0x003F 0x007F ]
VBIC         Q0, Q0, Q1   // Q0[i] = Q0[i] & (~Q1[i])  i={0..7} ;
                          // Clear elements of Q0 given by Q1, data type can be eluded
                          // Q2 = [ 0xFFFF 0xFFFE 0xFFFC 0xFFF8 0xFFF0 0xFFE0
                          // 0xFFC0 0xFF80 ]
```

3.30.6 VEOR

Vector Bitwise Exclusive Or.

Syntax

```
VEOR<v>{.<dt>} Qd, Qn, Qm
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Parameters

`Qd` Destination vector register.

`Qm` Source vector register.

`Qn` Source vector register.

`dt` An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following:

- `S8`
- `S16`
- `S32`
- `U8`
- `U16`
- `U32`
- `I8`
- `I16`
- `I32`
- `F16`
- `F32`

`v` See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise EOR of a vector register with another vector register. The result is written to the destination vector register.

VEOR example

```
// Exclusive Or between 8-bit incrementing and decrementing vectors
MOV            R0, #0              // Incrementing sequence start
VIDUP.U8       Q0, R0, #1          // Generator, increment step of 1
VDDUP.U8       Q1, R0, #1          // Generator, decrement step of 1, R0 = 16 after VIDUP
                                   // Q0 = [ 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08
0x09 0x0A 0x0B
                                   // 0x0C 0x0D 0x0E 0x0F]
                                   // Q1 = [ 0x10 0x0F 0x0E 0x0D 0x0C 0x0B 0x0A 0x09 0x08
0x07 0x06 0x05
                                   // 0x04 0x03 0x02 0x01]
VEOR           Q2, Q0, Q1          // Q2 = Q0 ^ Q1
                                   // Q2 = [ 0x10 0x0E 0x0C 0x0E 0x08 0x0E 0x0C 0x0E 0x00
0x0E 0x0C
                                   // 0x0E 0x08 0x0E 0x0C 0x0E]
```

3.30.7 VMOV (immediate)

Vector Move (immediate).

Syntax

```
VMOV<v>.<dt> Qd, #<imm>
```

Parameters

`Qd` Destination vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt
- Indicates the size of the elements in the vector.
- For use with the AdvSIMDExpandImm() function.
  The function takes an immediate value from the field or fields of an instruction encoding, or derived from the encoding and expands it to a 64-bit value. The input is an 8-bit value.
  This parameter must be one of the following values:
  — `I32`
  — `I16`
  — `I8`
  — `I64`
  — `F32`

imm The immediate value to load in to each element.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Set each element of a vector register to the immediate operand value.

VMOV (immediate) example

```
// Immediate vector move variants
VMOV.I32      Q0, #0x3F000000      // Q0[i] = 0x3F000000   i = {0..3}
                                   // Q0 = [ 0x3F000000 0x3F000000 0x3F000000 0x3F000000]

VMOV.F32      Q0, #-1.0e+00        // Q0[i] = -1.0f   i = {0..3}
                                   // Q0 = [ -1.0 -1.0 -1.0 -1.0 ]

VMOV.I16      Q0, #0x8             // Q0[i] = 8    i = {0..7}
                                   // Q0 = [ 8 8 8 8 8 8 8 8 ]
```

3.30.8 VMOV (register)

Vector Move (register).

Syntax

```
VMOV<v> Qd, Qm
is equivalent to
VORR<v> Qd, Qm, Qm
and is the preferred disassembly when <Qm == Qn>
```

Parameters

None.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Copy the value of one vector register to another vector register.

This is an alias of VORR with the following condition satisfied: `Qm == Qn`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

VMOV (register) example

```
// Vector move
MOV          R0, #0
VIDUP.U32    Q0, R0, #2    // Incrementing sequence, starting at 0, increment of 2
                           // Q0 = [ 0 2 4 6 ]

VMOV         Q1, Q0        // Q1 = Q0
                           // Q1 = [ 0 2 4 6 ]
```

3.30.9 VMOV (general-purpose register to vector lane)

Vector Move (general-purpose register to vector lane).

Syntax

```
VMOV<c>.<dt> Qd[idx], Rt
```

Parameters

Qd Destination vector register.

Rt Source general-purpose register.

c See Standard Assembler Syntax Fields

dt
- Indicates the size of the elements in the vector.
- This parameter must be one of the following values
  - 32
  - 16
  - 8

idx Element index to select in the vector register, must be in the range 0 to ((128/`dt`)-1). This value is encoded into the bits of h:op1:op2 which are not used to encode `dt`.

Restrictions

`Rt` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Copy the value of a general-purpose register to a vector lane.

VMOV (general-purpose register to vector lane) example

```
// Move General-purpose register to vector lane
MOV           R2, #0
VIDUP.U32     Q0, R2, #1            // Generates incrementing sequence,
                                    // starting at 0 with step of 1
                                    // Q0 = [ 0 1 2 3 ]
MOV           R0, #0
VMOV.32       Q0[2], R0             // Q0[2] = R0
                                    // Q0 = [ 0 1 0 3 ]
```

3.30.10 VMOV (vector lane to general-purpose register)

Vector Move (vector lane to general-purpose register).

Syntax

```
VMOV<c>.<dt> Rt, Qn[idx]
```

Parameters

Qn Source vector register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Rt Destination general-purpose register.

c See Standard Assembler Syntax Fields

dt This parameter determines the following values:

- S8
- U8
- S16
- U16
- S32
- U32

idx Element index to select in the vector register, must be in the range 0 to ((128/`dt`)-1). This value is encoded into the bits of `h:op1:op2` which are not used to encode `dt`.

Restrictions

`Rt` must not use the same register as `SP` and `PC`

Post-conditions

There are no condition flags.

Operation

Copy the value of a vector lane to a general-purpose register.

VMOV (vector lane to general-purpose register) example

```
// Move vector lane to general-purpose register
MOV             R2, #0
VIDUP.U32       Q0, R2, #1              // Generates incrementing sequence,
                                        // starting at 0 with step of 1
                                        // Q0 = [ 0 1 2 3 ]

VMOV.32         R0, Q0[3]               // R0 = Q0[3]
                                        // R0 = 3
```

3.30.11 VMVN (immediate)

Vector Bitwise NOT.

Syntax

```
VMVN<v>.<dt> Qd, #<imm>
```

Parameters

Qd Destination vector register.

dt
- Indicates the size of the elements in the vector.
- For use with the AdvSIMDExpandImm() function. The function takes an immediate value from the field or fields of an instruction encoding, or derived from the encoding and expands it to a 64-bit value. The input is an 8-bit value. This parameter must be one of the following values:
  - I32
  - I16

imm The immediate value to load in to each element.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

Set each element of a vector register to the bitwise inverse of the immediate operand value.

VMVN (immediate) example

```
// 16-bit integer Vector Bitwise without Immediate 8
VMVN.I16        Q0, #8    // Q0[i] = ~8   i={0..7}
                          // Q0 = [-9 -9 -9 -9 -9 -9 -9 -9]

// 32-bit integer Vector Bitwise NOT with Immediate 50
VMVN.I32        Q0, #50   // Q0[i] = ~50  i={0..3}
                          // Q0 = [ -51 -51 -51 -51 ]
```

3.30.12 VMVN (register)

Vector Bitwise Not.

Syntax

```
VMVN<v> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Bitwise invert the value of a vector register and place the result in another vector register.

VMVN (register) example

```
// 16-bit integer Vector Bitwise without register input
VMOV.S16       Q0, #8
                         // Q0 = [8 8 8 8 8 8 8 8]
VMVN           Q1, Q0    // Q1 = ~Q0
                         // Q1 = [-9 -9 -9 -9 -9 -9 -9 -9]
```

3.30.13 VORN

Vector Bitwise Or Not.

Syntax

```
VORN<v>{.<dt>} Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following:

- `S8`
- `S16`
- `S32`
- `U8`
- `U16`
- `U32`
- `I8`
- `I16`
- `I32`
- `F16`
- `F32`

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise OR NOT of a vector register with another vector register. The result is written to the destination vector register.

VORN example

```
// 32-bit Vector Bitwise Or Not
MOV           R2, #0
MOV           R3, #1000
VIDUP.U32     Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
VMOV.S32      Q1, #255
                            // Q0 = [0 1000 2000 3000]
                            // Q1 = [255 255 255 255]

VORN.S32      Q2, Q0, Q1    // Q2[i] = Q1[i] | ~Q0[i]   i={0..3}
                            // Q2 = [-256 -24 -48 -72]
                            // Q2 = [0xFFFFFF00 0xFFFFFFE8 0xFFFFFFD0 0xFFFFFFB8]
```

3.30.14 VORN (immediate)

Vector Bitwise OR NOT.

Syntax

```
VORN<v>.<dt> Qda, #<imm>
is equivalent to
VORR<v>.<dt> Qda, #~<imm>
and is never the preferred disassembly
```

Parameters

None.

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Operation

This is a pseudo-instruction, equivalent to a VORR (immediate) instruction with the immediate value bitwise inverted.

VORN (immediate) example

```
MOV          R2, #0
MOV          R3, #1000
VIDUP.U16    Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S16     Q0, Q0, R3    // Multiply by 1000

VORN.I16     Q0, #255      // Q0[i] = Q0[i] | ~0x00FF   i={0..7}
                           // Q0 = [0xFFE8 0xFF00 0xFFB8 0xFFD0 0xFF88 0xFFA0 0xFF58 0xFF70]
```

3.30.15 VORR

Vector Bitwise Or.

Syntax

```
VORR<v>{.<dt>} Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

dt An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following:

- S8
- S16
- S32
- U8
- U16
- U32
- I8
- I16
- I32
- F16
- F32

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bitwise OR of a vector register with another vector register. The result is written to the destination vector register.

VORR example

```
// 32-bit Vector Bitwise Or
MOV          R2, #0
MOV          R3, #1000
VIDUP.U32    Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S32     Q0, Q0, R3    // Multiply by 1000
VMOV.S32     Q1, #255
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                            // Q0 = [0 1000 2000 3000]
                            // Q1 = [255 255 255 255]

VORR          Q2, Q0, Q1    // Q2[i] = Q1[i] | Q0[i]   i={0..3}
                            // Q2 = [255 1023 2047 3071]
```

3.30.16 VORR (immediate)

Vector Bitwise OR.

Syntax

```
VORR<v>.<dt> Qda, #<imm>
```

Parameters

Qda Source and destination vector register.

dt Indicates the size of the elements in the vector. This parameter must be one of the following values:

- I32
- I16

imm The immediate value to load in to each element.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

OR the value of a vector register with the immediate operand value.

VORR (immediate) example

```
// 32-bit Vector Bitwise Or with immediate value
MOV           R2, #0
MOV           R3, #1000
VIDUP.U32     Q0, R2, #1    // Generates incrementing sequence, starting at 0 with step of 1
VMUL.S32      Q0, Q0, R3    // Multiply by 1000
                            // Q0 = [0 1000 2000 3000]
                            // Q1 = [255 255 255 255]

VORR.I32      Q0, #0xF      // Q0[i] = Q0[i] | 0xF  i={0..3}
                            // Q2 = [15 1007 2015 3007]
```

3.30.17 VPSEL

Vector Predicated Select.

Syntax

```
VPSEL<v>{.<dt>} Qd, Qn, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Qn Source vector register.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

dt An optional data type. It is ignored by assemblers and does not affect the encoding. This can be one of the following:

- S8.
- S16.
- S32.
- U8.
- U16.
- U32.
- I8.
- i16.
- I32.
- F16.
- F32.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Compute a bytewise conditional select of a vector register with another vector register, based on the VPR predicate bits.

VPSEL example

```
// Vector Predicated Select
// 32-bit vector selection based on comparison
MOV             R0, #0
VIDUP.U32       Q0, R0, #1   // 32-bit Generator, increment step of 1
VDDUP.U32       Q1, R0, #1   // 32-bit Generator, decrement step of 1
VMOV.S32        Q2, #10      // Q2[i] = 10        i={0..3}
VMOV.S32        q3, #20      // Q3[i] = 20        i={0..3}
VCMP.S32        LE, Q0, Q1   // Q0[i] <= Q1[i] ?  i={0..3}, set VPR.P0
                             // Q0 = [ 0 1 2 3 ]
                             // Q1 = [ 4 3 2 1 ]
VPSEL           Q2, Q2, Q3   // Performs selection based on comparison
                             // Q2[i] = Q0[i] <= Q1[i] ? Q2[i] : Q3[i]     i={0..3}
                             // Q2 = [ 10 10 10 20  ]
```

3.30.18 VREV16

Vector Reverse.

Syntax

```
VREV16<v>.<size> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

Size Indicates the size of the elements in the vector. This parameter must 8.

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Post-conditions

There are no condition flags.

Operation

Reverse the order of 8-bit elements within each halfword of the source vector register and places the result in the destination vector register.

VREV16 example

```
// Vector Reverse. Reverse the order of elements within each halfword
MOV         R0, #1
VIDUP.U8    Q0,R0,#1    // Incrementing sequence, starting at 1, increment of 1

                        // Q0 = [  1 2 3 4  5 6 7 8  9 10 11 12  13 14 15 16 ]
VREV16.8    Q1,Q0       // Q1[2*i] = Q0[2*i+1]   i={0..7}
                        // Q1[2*i+1] = Q0[2*i]   i={0..7}
                        // Q1 = [  2 1 4 3  6 5 8 7  10 9 12 11  14 13 16 15 ]
```

3.30.19 VREV32

Vector Reverse.

Syntax

```
VREV32<v>.<size> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

size Indicates the size of the elements in the vector.This parameter must be one of the following values:

- 8
- 16

v See Standard Assembler Syntax Fields

Restrictions

There are no restrictions.

Post-conditions

There are no condition flags.

Operation

Reverse the order of 8-bit or 16-bit elements within each word of the source vector register and places the result in the destination vector register.

VREV32 example

```
// Vector Reverse. Reverse the order of elements within each word
MOV         R0, #1
VIDUP.U8    Q0,R0,#1    // Incrementing sequence, starting at 1, increment of 1

                        // Q0 = [  1 2 3 4  5 6 7 8  9 10 11 12  13 14 15 16 ]
VREV32.8    Q1,Q0       // Q1[4*i] = Q0[4*i+3]        i = {0..3}
                        // Q1[4*i+1] = Q0[4*i+2]
                        // Q1[4*i+2] = Q0[4*i+1]
                        // Q1[4*i+3] = Q0[4*i]
                        // Q1 = [  4 3 2 1  8 7 6 5  12 11 10 9  16 15 14 13 ]

MOV         R0, #1
VIDUP.U16   Q0,R0,#1    // Incrementing sequence, starting at 1, increment of 1

                        // Q0 = [  1 2  3 4  5 6  7 8  ]
VREV32.16   Q1,Q0       // Q1[2*i] = Q0[2*i+1]     i = {0..3}
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
                        // Q1[2*i+1] = Q0[2*i]
                        // Q1 = [  2 1  4 3  6 5  8 7  ]
```

3.30.20 VREV64

Vector Reverse.

Syntax

```
VREV64<v>.<size> Qd, Qm
```

Parameters

Qd Destination vector register.

Qm Source vector register.

size Indicates the size of the elements in the vector. This parameter must be one of the following values
- 8
- 16
- 32

v See Standard Assembler Syntax Fields

Restrictions

Qd must not use the same register as Qm

Post-conditions

There are no condition flags.

Operation

Reverse the order of 8-bit, 16-bit or 32-bit elements within each doubleword of the source vector register and places the result in the destination vector register.

VREV64 example

```
// Vector Reverse. Reverse the order of elements within each doubleword
MOV         R0, #1
VIDUP.U8    Q0, R0, #1  // Incrementing sequence, starting at 1, increment of 1
                        // Q0 = [  1 2 3 4  5 6 7 8  9 10 11 12  13 14 15 16 ]
// Reverse bytes within doubleword
VREV64.8    Q1,Q0       // Q1[8*i] = Q0[8*i+7]  i = {0..1}
                        // Q1[8*i+1] = Q0[8*i+6]
                        // Q1[8*i+2] = Q0[8*i+5]
                        // ...
                        // Q1[8*i+7] = Q0[8*i]
                        // Q1 = [  8 7 6 5  4 3 2 1  16 15 14 13  12 11 10 9 ]


MOV         R0, #1
VIDUP.U16   Q0, R0, #1  // Incrementing sequence, starting at 1, increment of 1
                        // Q0 = [  1 2  3 4  5 6  7 8  ]
// Reverse 16-bit short within doubleword
VREV64.16   Q1, Q0      // Q1[4*i] = Q0[4*i+3]    i = {0..1}
                        // Q1[4*i+1] = Q0[4*i+2]
                        // Q1[4*i+2] = Q0[4*i+1]
                        // Q1[4*i+3] = Q0[4*i]
                        // Q1 = [  4 3  2 1  8 7  6 5  ]

MOV           R0, #1
VIDUP.U16     Q0, R0, #1 // Incrementing sequence, starting at 1, increment of 1
VCVT.F16.S16  Q1, Q0     // Convert into F16 vector
                         // Q1 = [ 1.0 2.0  3.0 4.0  5.0 6.0  7.0 8.0 ]
// Reverse half float within doubleword
VREV64.16     Q2, Q1     // Q2[4*i] = Q1[4*i+3]   i = {0..1}
                         // Q2[4*i+1] = Q1[4*i+2]
                         // Q2[4*i+2] = Q1[4*i+1]
                         // Q2[4*i+3] = Q1[4*i]
                         // Q2 = [ 4.0 3.0  2.0 1.0  8.0 7.0  6.0 5.0 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
MOV         R0, #1
VIDUP.U32   Q0, R0, #1  // Incrementing sequence, starting at 1, increment of 1
                        // Q0 = [ 1 2 3 4 ]
// Reverse 32-bit words within doubleword
VREV64.32   Q1, Q0      // Q1[2*i] = Q0[2*i+1]    i = {0..1}
                        // Q1[2*i+1] = Q0[2*i]
                        // Q1 = [ 2 1 4 3 ]


MOV           R0, #1
VIDUP.u32     Q0, R0, #1 // Incrementing sequence, starting at 1, increment of 1
VCVT.F32.S32  Q1, Q0     // Convert into F32 vector
                         // Q1 = [ 1.0 2.0 3.0 4.0 ]
// Reverse 32-bit float within doubleword
VREV64.32     Q2, Q1     // Q2[2*i] = Q1[2*i+1]  i = {0..1}
                         // Q2[2*i+1] = Q1[2*i]
                         // Q2 = [ 2.0 1.0 4.0 3.0 ]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31 Miscellaneous instructions

Reference material for the Cortex-M55 processor miscellaneous instructions.

3.31.1 List of miscellaneous instructions

An alphabetically ordered list of the miscellaneous instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-24 Miscellaneous instructions

Mnemonic	Brief description	See
`BKPT`	Breakpoint	*3.31.2 BKPT* on page 3-426
`CPSID`	Change Processor State, Disable Interrupts	*3.31.3 CPS* on page 3-427
`CPSIE`	Change Processor State, Enable Interrupts	*3.31.3 CPS* on page 3-427
`DMB`	Data Memory Barrier	*3.31.5 DMB* on page 3-428
`DSB`	Data Synchronization Barrier	*3.31.6 DSB* on page 3-429
`ISB`	Instruction Synchronization Barrier	*3.31.7 ISB* on page 3-430
`MRS`	Move from special register to register	*3.31.8 MRS* on page 3-431
`MSR`	Move from register to special register	*3.31.9 MSR* on page 3-432
`NOP`	No Operation	*3.31.10 NOP* on page 3-433
`SEV`	Send Event	*3.31.11 SEV* on page 3-434
`SG`	Secure Gateway	*3.31.12 SG* on page 3-435
`SVC`	Supervisor Call	*3.31.13 SVC* on page 3-436
`TT`	Test Target	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
`TTT`	Test Target Unprivileged	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
`TTA`	Test Target Alternate Domain	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
`TTAT`	Test Target Alternate Domain Unprivileged	*3.31.14 TT, TTT, TTA, and TTAT* on page 3-437
`WFE`	Wait For Event	*3.31.16 WFE* on page 3-440
`WFI`	Wait For Interrupt	*3.31.17 WFI* on page 3-441
`YIELD`	Yield	*3.31.18 YIELD* on page 3-441

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.2 BKPT

Breakpoint.

Syntax

`BKPT #imm`

Where:

imm Is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The `BKPT` instruction causes the processor to enter Debug state if invasive debug is enabled. Debug tools can use this to investigate system state when the instruction at a particular address is reached.

imm is ignored by the processor. If required, a debugger can use it to store additional information about the breakpoint.

The `BKPT` instruction can be placed inside an IT block, but it executes unconditionally, unaffected by the condition specified by the `IT` instruction.

Condition flags

This instruction does not change the flags.

Example 3-55 Examples

```
BKPT #0x3   ; Breakpoint with immediate value set to 0x3 (debugger can
            ; extract the immediate value by locating it using the PC)
```

Note

Arm does not recommend the use of the `BKPT` instruction with an immediate value set to `0xAB` for any purpose other than Semi-hosting.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.3 CPS

Change Processor State.

Syntax

`CPS`*`effect iflags`*

Where:

effect	Is one of:	
	`IE`	Clears the special purpose register.
	`ID`	Sets the special purpose register.
iflags	Is a sequence of one or more flags:	
	`i`	Set or clear PRIMASK.
	`f`	Set or clear FAULTMASK.

Operation

`CPS` changes the PRIMASK and FAULTMASK special register values.

Restrictions

The restrictions are:
- Use `CPS` only from privileged software. It has no effect if used in unprivileged software.
- `CPS` cannot be conditional and so must not be used inside an IT block.

Condition flags

This instruction does not change the condition flags.

Example 3-56 Examples

```
CPSID i  ; Disable interrupts and configurable fault handlers (set PRIMASK)
CPSID f  ; Disable interrupts and all fault handlers (set FAULTMASK)
CPSIE i  ; Enable interrupts and configurable fault handlers (clear PRIMASK)
CPSIE f  ; Enable interrupts and fault handlers (clear FAULTMASK)
```

3.31.4 CPY

Copy is a pre-Unified Assembler Language (UAL) synonym for MOV (register).

Syntax

`CPY` *`Rd, Rn`*

This is equivalent to:

`MOV` *`Rd, Rn`*

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.5 DMB

Data Memory Barrier.

Syntax

`DMB{cond} {opt}`

Where:

cond
: Is an optional condition code.

opt
: Specifies an optional limitation on the DMB operation. Values are:

SY
: DMB operation ensures ordering of all accesses, encoded as *opt* == '1111'. Can be omitted.

All other encodings of *opt* are RESERVED. The corresponding instructions execute as system (SY) DMB operations, but software must not rely on this behavior.

Operation

DMB acts as a data memory barrier. It ensures that all explicit memory accesses that appear, in program order, before the DMB instruction are completed before any explicit memory accesses that appear, in program order, after the DMB instruction. DMB does not affect the ordering or execution of instructions that do not access memory.

Condition flags

This instruction does not change the flags.

Example 3-57 Examples

```
DMB  ; Data Memory Barrier
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.6 DSB

Data Synchronization Barrier.

Syntax

`DSB{cond} {opt}`

Where:

cond
: Is an optional condition code.

opt
: Specifies an optional limitation on the DSB operation. Values are:

SY
: DSB operation ensures completion of all accesses, encoded as *opt* == '1111'. Can be omitted.

All other encodings of *opt* are RESERVED. The corresponding instructions execute as system (SY) DSB operations, but software must not rely on this behavior.

Operation

DSB acts as a special data synchronization memory barrier. Instructions that come after the DSB, in program order, do not execute until the DSB instruction completes. The DSB instruction completes when all explicit memory accesses before it complete.

Condition flags

This instruction does not change the flags.

Example 3-58 Examples

```
DSB ; Data Synchronisation Barrier
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.7 ISB

Instruction Synchronization Barrier.

Syntax

`ISB{cond} {opt}`

Where:

cond
: Is an optional condition code.

opt
: Specifies an optional limitation on the ISB operation. Values are:

SY
: Fully system ISB operation, encoded as *opt* == '1111'. Can be omitted.

All other encodings of *opt* are RESERVED. The corresponding instructions execute as full system ISB operations, but software must not rely on this behavior.

Operation

`ISB` acts as an instruction synchronization barrier. It flushes the pipeline of the processor, so that all instructions following the `ISB` are fetched from cache or memory again, after the `ISB` instruction has been completed.

Condition flags

This instruction does not change the flags.

Example 3-59 Examples

```
ISB  ; Instruction Synchronisation Barrier
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.8 MRS

Move the contents of a special register to a general-purpose register.

Syntax

`MRS{cond} Rd, spec_reg`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
spec_reg	Can be any of: `APSR`, `IPSR`, `EPSR`, `IEPSR`, `IAPSR`, `EAPSR`, `PSR`, `MSP`, `PSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, `CONTROL`,`MSP_NS`, `PSP_NS`, `MSPLIM`, `PSPLIM`, `MSPLIM_NS`, `PSPLIM_NS`, `PRIMASK_NS`, `FAULTMASK_NS`, and `CONTROL_NS`. **Note** All the `EPSR` and `IPSR` fields are zero when read by the `MRS` instruction. 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.

Operation

Use `MRS` in combination with `MSR` as part of a read-modify-write sequence for updating a PSR, for example to clear the Q flag.

In process swap code, the programmers model state of the process being swapped out must be saved, including relevant PSR contents. Similarly, the state of the process being swapped in must also be restored. These operations use `MRS` in the state-saving instruction sequence and `MSR` in the state-restoring instruction sequence.

Note

`BASEPRI_MAX` is an alias of `BASEPRI` when used with the `MRS` instruction.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Example 3-60 Examples

```
MRS  R0, PRIMASK ; Read PRIMASK value and write it to R0
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.9 MSR

Move the contents of a general-purpose register into the specified special register.

Syntax

`MSR{cond} spec_reg, Rn`

Where:

cond Is an optional condition code.

Rn Is the source register.

spec_reg Can be any of: `APSR_nzcvq`, `APSR_g`, `APSR_nzcvqg`, `MSP`, `PSP`, `PRIMASK`, `BASEPRI`, `BASEPRI_MAX`, `FAULTMASK`, `CONTROL`, `MSP_NS`, `PSP_NS` -`MSPLIM`, `PSPLIM`, `MSPLIM_NS`, `PSPLIM_NS`, `PRIMASK_NS`, `FAULTMASK_NS`, and `CONTROL_NS`.

Note

You can use `APSR` to refer to `APSR_nzcvq`.

Operation

The register access operation in `MSR` depends on the privilege level. Unprivileged software can only access the `APSR`, see the APSR bit assignments. Privileged software can access all special registers.

In unprivileged software writes to unallocated or execution state bits in the `PSR` are ignored.

Note

When you write to `BASEPRI_MAX`, the instruction writes to `BASEPRI` only if either:
- *Rn* is non-zero and the current `BASEPRI` value is 0.
- *Rn* is non-zero and less than the current `BASEPRI` value.

Note

An access to a register not ending in _NS writes the register associated with the current Security state. Access to a register ending in _NS in Secure state writes the Non-secure register. Access to a register ending in _NS in Non-secure state is RAZ/WI.

Restrictions

Rn must not be SP and must not be PC.

Condition flags

This instruction updates the flags explicitly based on the value in *Rn*.

Example 3-61 Examples

```
MSR  CONTROL, R1 ; Read R1 value and write it to the CONTROL register.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.10 NOP

No Operation.

Syntax

`NOP{cond}`

Where:

cond	Is an optional condition code.

Operation

`NOP` does nothing. `NOP` is not necessarily a time-consuming `NOP`. The processor might remove it from the pipeline before it reaches the execution stage.

Use `NOP` for padding, for example to place the following instruction on a 64-bit boundary.

Condition flags

This instruction does not change the flags.

Example 3-62 Examples

```
NOP  ; No operation
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.11 SEV

Send Event.

Syntax

`SEV{cond}`

Where:

cond	Is an optional condition code.

Operation

`SEV` is a hint instruction that causes an event to be signaled to all processors within a multiprocessor system. It also sets the local event register to 1.

Condition flags

This instruction does not change the flags.

Example 3-63 Examples

```
SEV ; Send Event
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.12 SG

Secure Gateway.

Syntax

`SG`

Operation

Secure Gateway marks a valid branch target for branches from Non-secure code that wants to call Secure code.

A linker is expected to generate a Secure Gateway operation as a part of the branch table for the *Non-secure Callable* (NSC) region.

There is no C intrinsic function for `SG`. Secure Gateways are expected to be generated by linker or by assembly programming. Arm does not expect software developers to insert a Secure Gateway instruction inside C or C++ program code.

Note

For information about how to build a Secure image that uses a previously generated import library, see the *Arm® Compiler Software Development Guide*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.13 SVC

Supervisor Call.

Syntax

`SVC{cond} #imm`

Where:

cond	Is an optional condition code.
imm	Is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The `SVC` instruction causes the `SVC` exception.

imm is ignored by the processor. If required, it can be retrieved by the exception handler to determine what service is being requested.

Condition flags

This instruction does not change the flags.

Example 3-64 Examples

```
SVC  #0x32     ; Supervisor Call (SVCall handler can extract the immediate value
               ; by locating it through the stacked PC)
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.14 TT, TTT, TTA, and TTAT

Test Target (Alternate Domain, Unprivileged).

Syntax

`{op}{cond} Rd, Rn`

Where:

op Is one of:

- `TT` *Test Target* (TT) queries the Security state and access permissions of a memory location.
- `TTT` *Test Target Unprivileged* (TTT) queries the Security state and access permissions of a memory location for an unprivileged access to that location.
- `TTA` In an implementation with the Security Extension, *Test Target Alternate Domain* (TTA) queries 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.
- `TTAT` In an implementation with the Security Extension, *Test Target Alternate Domain Unprivileged* (TTAT) queries the Security state and access permissions of a memory location for a Non-secure and unprivileged access to that location. These instructions are only valid when executing in Secure state, and are UNDEFINED if used from Non-secure state.

cond Is an optional condition code.

Rd Is the destination general-purpose register into which the status result of the target test is written.

Rn Is the base register.

Operation

The instruction returns the Security state and access permissions in the destination register, the contents of which are as follows:

Table 3-25 Security state and access permissions in the destination register

Bits	Name	Description
[7:0]	MREGION	The MPU region that the address maps to. This field is 0 if MRVALID is 0.
[15:8]	SREGION	In an implementation without the Security Extension, this field is RAZ/WI. The SAU region that the address maps to. This field is only valid if the instruction is executed from Secure state. This field is 0 if SRVALID is 0.
[16]	MRVALID	Set to 1 if the MREGION content is valid. Set to 0 if the MREGION content is invalid.
[17]	SRVALID	In an implementation without the Security Extension, this field is RAZ/WI. Set to 1 if the SREGION content is valid. Set to 0 if the SREGION content is invalid.
[18]	R	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 bit returns the permissions for unprivileged access, regardless of whether the current mode is privileged or unprivileged.
[19]	RW	Read/write accessibility. 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.
[31:20]	-	RAZ/WI
[20]	NSR	Equal to R AND NOT S. Can be used with the LSLS (immediate) instruction to check both the MPU and SAU or IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the R field is valid.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 3-25 Security state and access permissions in the destination register (continued)

Bits	Name	Description
[21]	NSRW	Equal to RW AND NOT S. Can be used with the LSLS (immediate) instruction to check both the MPU and SAU or IDAU permissions. This bit is only valid if the instruction is executed from Secure state and the RW field is valid.
[22]	S	Security. A value of 1 indicates that the memory location is Secure, and a value of 0 indicates that the memory location is Non-secure. This bit is only valid if the instruction is executed from Secure state.
[23]	IRVALID	IREGION valid flag. For a Secure request, indicates the validity of the IREGION field. Set to 1 if the IREGION content is valid. Set to 0 if the IREGION content is invalid. This bit is always 0 if the IDAU cannot provide a region number, the address is exempt from security attribution, or if the requesting TT instruction is executed from the Non-secure state.
[31:24]	IREGION	IDAU region number. Indicates the IDAU region number containing the target address. This field is 0 if IRVALID is 0.

Invalid fields are 0.

The MREGION field is invalid and 0 if any of the following conditions are true:

- The MPU is not present or MPU_CTRL.ENABLE is 0.
- The address did not match any enabled MPU regions.
- The address matched multiple MPU regions.
- TT was executed from an unprivileged mode, or TTA is executed and Non-secure state is unprivileged.

The R, RW, NSR, and NSRW bits are invalid and 0 if any of the following conditions are true:

- The address matched multiple MPU regions.
- TT is executed from an unprivileged mode, or TTA is executed and Non-secure state is unprivileged.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.15 UDF

Permanently Undefined.

Syntax

```
UDF{cond}.W {#}imm
```

Where:

imm Is a:

- 8-bit unsigned immediate, in the range 0 to 255. The processor ignores the value of this constant.
- 16-bit unsigned immediate, in the range 0 to 65535. The processor ignores the value of this constant.

cond Arm deprecates using any *c* value other than `AL`.

Operation

Permanently Undefined generates an Undefined Instruction UsageFault exception.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.16 WFE

Wait For Event.

Syntax

`WFE{cond}`

Where:

cond Is an optional condition code.

Operation

`WFE` is a hint instruction.

If the event register is 0, `WFE` suspends execution until one of the following events occurs:

- An exception, unless masked by the exception mask registers or the current priority level.
- An exception enters the Pending state, if `SEVONPEND` in the System Control Register is set.
- A Debug Entry request, if Debug is enabled.
- An event signaled by a peripheral or another processor in a multiprocessor system using the `SEV` instruction.

If the event register is 1, `WFE` clears it to 0 and returns immediately.

Condition flags

This instruction does not change the flags.

Example 3-65 Examples

```
WFE  ; Wait for event
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.31.17 WFI

Wait for Interrupt.

Syntax

`WFI{cond}`

Where:

cond	Is an optional condition code.

Operation

`WFI` is a hint instruction that suspends execution until one of the following events occurs:

- A non-masked interrupt occurs and is taken.
- An interrupt masked by PRIMASK becomes pending.
- A Debug Entry request, if Debug is enabled.

Condition flags

This instruction does not change the flags.

Example 3-66 Examples

```
WFI ; Wait for interrupt
```

3.31.18 YIELD

Yield

Syntax

`YIELD{cond}`

Where:

cond	Is an optional condition code.

Operation

`YIELD` is a hint instruction that enables software with a multithreading capability to indicate to the hardware that a task is being performed, which 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.

Condition flags

This instruction does not change the flags.

Example 3-67 Examples

```
YIELD; Suspend task
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32 Memory access instructions

Reference material for the Cortex-M55 processor memory access instruction set.

3.32.1 List of memory access instructions

An alphabetically ordered list of the memory access instructions, with a brief description and link to the syntax definition, operations, restrictions, and example usage for each instruction.

Table 3-26 Memory access instructions

Mnemonic	Brief description	See
`ADR`	Generate PC-relative address	*3.32.2 ADR* on page 3-443
`CLREX`	Clear Exclusive	*3.32.13 CLREX* on page 3-463
`LDM{mode}`	Load Multiple registers	*3.32.7 LDM and STM* on page 3-453
`LDA{type}`	Load-Acquire	*3.32.10 LDA and STL* on page 3-458
`LDAEX`	Load-Acquire Exclusive	*3.32.12 LDAEX and STLEX* on page 3-461
`LDR{type}`	Load Register using immediate offset	*3.32.3 LDR and STR, immediate offset* on page 3-444
`LDR{type}`	Load Register using register offset	*3.32.4 LDR and STR, register offset* on page 3-447
`LDR{type}T`	Load Register with unprivileged access	*3.32.5 LDR and STR, unprivileged* on page 3-449
`LDR`	Load Register using PC-relative address	*3.32.6 LDR, PC-relative* on page 3-451
`LDRD`	Load Register Dual	*3.32.3 LDR and STR, immediate offset* on page 3-444
`LDREX{type}`	Load Register Exclusive	*3.32.11 LDREX and STREX* on page 3-459
`PLD`	Preload Data.	*3.32.8 PLD* on page 3-455
`POP`	Pop registers from stack	*3.32.9 PUSH and POP* on page 3-456
`PUSH`	Push registers onto stack	*3.32.9 PUSH and POP* on page 3-456
`STL{mode}`	Store-Release	*3.32.10 LDA and STL* on page 3-458
`STLEX`	Store Release Exclusive	*3.32.12 LDAEX and STLEX* on page 3-461
`STM{mode}`	Store Multiple registers	*3.32.7 LDM and STM* on page 3-453
`STR{type}`	Store Register using immediate offset	*3.32.3 LDR and STR, immediate offset* on page 3-444
`STR{type}`	Store Register using register offset	*3.32.4 LDR and STR, register offset* on page 3-447
`STR{type}T`	Store Register with unprivileged access	*3.32.5 LDR and STR, unprivileged* on page 3-449
`STREX{type}`	Store Register Exclusive	*3.32.11 LDREX and STREX* on page 3-459

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.2 ADR

Generate PC-relative address.

Syntax

`ADR{cond} Rd, label`

Where:

cond	Is an optional condition code.
Rd	Is the destination register.
label	Is a PC-relative expression.

Operation

`ADR` generates an address by adding an immediate value to the PC, and writes the result to the destination register.

`ADR` provides the means by which position-independent code can be generated, because the address is PC-relative.

If you use `ADR` to generate a target address for a `BX` or `BLX` instruction, you must ensure that bit[0] of the address you generate is set to1 for correct execution.

Values of *label* must be within the range of −4095 to +4095 from the address in the PC.

Note

You might have to use the `.W` suffix to get the maximum offset range or to generate addresses that are not word-aligned.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Example 3-68 Examples

```
ADR     R1, TextMessage   ; Write address value of a location labelled as
                          ; TextMessage to R1.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.3 LDR and STR, immediate offset

Load and Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}] ; immediate offset
op{type}{cond} Rt, [Rn, #offset]! ; pre-indexed
op{type}{cond} Rt, [Rn], #offset ; post-indexed
opD{cond} Rt, Rt2, [Rn {, #offset}] ; immediate offset, two words
opD{cond} Rt, Rt2, [Rn, #offset]! ; pre-indexed, two words
opD{cond} Rt, Rt2, [Rn], #offset ; post-indexed, two words
```

Where:

op
Is one of:

LDR
Load Register.

STR
Store Register.

type
Is one of:

B	Unsigned byte, zero extend to 32 bits on loads.
SB	Signed byte, sign extend to 32 bits (LDR only).
H	Unsigned halfword, zero extend to 32 bits on loads.
SH	Signed halfword, sign extend to 32 bits (LDR only).
-	Omit, for word.

cond
Is an optional condition code.

Rt
Is the register to load or store.

Rn
Is the register on which the memory address is based.

offset
Is an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*.

Rt2
Is the additional register to load or store for two-word operations.

Operation

LDR instructions load one or two registers with a value from memory.

STR instructions store one or two register values to memory.

Load and store instructions with immediate offset can use the following addressing modes:

Offset addressing
The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access. The register *Rn* is unaltered. The assembly language syntax for this mode is:

```
[Rn, #offset]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Pre-indexed addressing

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access and written back into the register *Rn*. The assembly language syntax for this mode is:

```
[Rn, #offset]!
```

Post-indexed addressing

The address obtained from the register *Rn* is used as the address for the memory access. The offset value is added to or subtracted from the address, and written back into the register *Rn*. The assembly language syntax for this mode is:

```
[Rn], #offset
```

The value to load or store can be a byte, halfword, word, or two words. Bytes and halfwords can either be signed or unsigned.

The following table shows the ranges of offset for immediate, pre-indexed and post-indexed forms.

Table 3-27 Offset ranges

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	−255 to 4095	−255 to 255	−255 to 255
Two words	multiple of 4 in the range −1020 to 1020	multiple of 4 in the range −1020 to 1020	multiple of 4 in the range −1020 to 1020

Restrictions

For load instructions:

- *Rt* can be SP or PC for word loads only.
- *Rt* must be different from *Rt2* for two-word loads.
- *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms.

When *Rt* is PC in a word load instruction:

- Bit[0] of the loaded value must be 1 for correct execution.
- A branch occurs to the address created by changing bit[0] of the loaded value to 0.
- If the instruction is conditional, it must be the last instruction in the IT block.

For store instructions:

- *Rt* can be SP for word stores only.
- *Rt* must not be PC.
- *Rn* must not be PC.
- *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms.

Condition flags

These instructions do not change the flags.

Example 3-69 Examples

```
LDR     R8, [R10]              ; Loads R8 from the address in R10.
LDRNE   R2, [R5, #960]!        ; Loads (conditionally) R2 from a word
                               ; 960 bytes above the address in R5, and
                               ; increments R5 by 960.
STR     R2, [R9,#const-struc]  ; const-struc is an expression evaluating
                               ; to a constant in the range 0-4095.
STRH    R3, [R4], #4           ; Store R3 as halfword data into address in
                               ; R4, then increment R4 by 4.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDRD    R8, R9, [R3, #0x20]    ; Load R8 from a word 32 bytes above the
                               ; address in R3, and load R9 from a word 36
                               ; bytes above the address in R3.
STRD    R0, R1, [R8], #-16     ; Store R0 to address in R8, and store R1 to
                               ; a word 4 bytes above the address in R8,
                               ; and then decrement R8 by 16.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.4 LDR and STR, register offset

Load and Store with register offset.

Syntax

op{*type*}{*cond*} *Rt*, [*Rn*, *Rm* {, LSL #*n*}]

Where:

op	Is one of: **LDR** Load Register. **STR** Store Register.
type	Is one of: **B** Unsigned byte, zero extend to 32 bits on loads. **SB** Signed byte, sign extend to 32 bits (LDR only). **H** Unsigned halfword, zero extend to 32 bits on loads. **SH** Signed halfword, sign extend to 32 bits (LDR only). **-** omit, for word.
cond	Is an optional condition code.
Rt	Is the register to load or store.
Rn	Is the register on which the memory address is based.
Rm	Is a register containing a value to be used as the offset.
LSL #*n*	Is an optional shift, with *n* in the range 0-3.

Operation

LDR instructions load a register with a value from memory.

STR instructions store a register value into memory.

The memory address to load from or store to is at an offset from the register *Rn*. The offset is specified by the register *Rm* and can be shifted left by up to 3 bits using LSL.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned.

Restrictions

In these instructions:

- *Rn* must not be PC.
- *Rm* must not be SP and must not be PC.
- *Rt* can be SP only for word loads and word stores.
- *Rt* can be PC only for word loads.

When *Rt* is PC in a word load instruction:

- Bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Condition flags

These instructions do not change the flags.

Example 3-70 Examples

```
STR    R0, [R5, R1]        ; Store value of R0 into an address equal to
                           ; sum of R5 and R1.
LDRSB  R0, [R5, R1, LSL #1] ; Read byte value from an address equal to
                           ; sum of R5 and two times R1, sign extended it
                           ; to a word value and put it in R0.
STR    R0, [R1, R2, LSL #2] ; Stores R0 to an address equal to sum of R1
                           ; and four times R2.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.5 LDR and STR, unprivileged

Load and Store with unprivileged access.

Syntax

`op{type}T{cond} Rt, [Rn {, #offset}]`

Where:

op Is one of:

LDR
Load Register.

STR
Store Register.

type Is one of:

B	Unsigned byte, zero extend to 32 bits on loads.
SB	Signed byte, sign extend to 32 bits (`LDR` only).
H	Unsigned halfword, zero extend to 32 bits on loads.
SH	Signed halfword, sign extend to 32 bits (`LDR` only).
-	Omit, for word.

cond Is an optional condition code.

Rt Is the register to load or store.

Rn Is the register on which the memory address is based.

offset Is an immediate offset from *Rn* and can be 0 to 255. If *offset* is omitted, the address is the value in *Rn*.

Operation

These load and store instructions perform the same function as the memory access instructions with immediate offset. The difference is that these instructions have only unprivileged access even when used in privileged software.

When used in unprivileged software, these instructions behave in exactly the same way as normal memory access instructions with immediate offset.

Restrictions

In these instructions:

- *Rn* must not be PC.
- *Rt* must not be SP and must not be PC.

Condition flags

These instructions do not change the flags.

Example 3-71 Examples

```
STRBTEQ  R4, [R7]       ; Conditionally store least significant byte in
                        ; R4 to an address in R7, with unprivileged access.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
LDRHT    R2, [R2, #8]   ; Load halfword value from an address equal to
                        ; sum of R2 and 8 into R2, with unprivileged access.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.6 LDR, PC-relative

Load register from memory.

Syntax

`LDR{type}{cond} Rt, label`

`LDRD{cond} Rt, Rt2, label ; Load two words`

Where:

type
Is one of:

B
Unsigned byte, zero extend to 32 bits.

SB
Signed byte, sign extend to 32 bits.

H
Unsigned halfword, zero extend to 32 bits.

SH
Signed halfword, sign extend to 32 bits.

-
Omit, for word.

cond
Is an optional condition code.

Rt
Is the register to load or store.

Rt2
Is the second register to load or store.

label
Is a PC-relative expression.

Operation

`LDR` loads a register with a value from a PC-relative memory address. The memory address is specified by a label or by an offset from the PC.

The value to load or store can be a byte, halfword, or word. For load instructions, bytes and halfwords can either be signed or unsigned.

label must be within a limited range of the current instruction. The following table shows the possible offsets between *label* and the PC.

Table 3-28 Offset ranges

Instruction type	Offset range
Word, halfword, signed halfword, byte, signed byte	−4095 to 4095
Two words	−1020 to 1020

Note

You might have to use the `.W` suffix to get the maximum offset range.

Restrictions

In these instructions:

- *Rt* can be SP or PC only for word loads.
- *Rt2* must not be SP and must not be PC.
- *Rt* must be different from *Rt2*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

When *Rt* is PC in a word load instruction:

- Bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Example 3-72 Examples

```
LDR     R0, LookUpTable   ; Load R0 with a word of data from an address
                          ; labelled as LookUpTable.
LDRSB   R7, localdata     ; Load a byte value from an address labelled
                          ; as localdata, sign extend it to a word
                          ; value, and put it in R7.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.7 LDM and STM

Load and Store Multiple registers.

Syntax

op{addr_mode}{cond} Rn{!}, reglist

Where:

op Is one of the following:

LDM
Load Multiple registers

STM
Store Multiple registers

addr_mode Is any one of the following:

IA
Increment address after each access. This is the default.

DB
Decrement address before each access.

cond Is an optional condition code.

Rn Is the register on which the memory addresses are based.

! Is an optional write-back suffix. If ! is present the final address, that is loaded from or stored to, is written back into *Rn*.

reglist Is a list of one or more registers to be loaded or stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

LDMIA and LDMFD are synonyms for LDM. LDMFD refers to its use for popping data from Full Descending stacks.

LDMEA is a synonym for LDMDB, and refers to its use for popping data from Empty Ascending stacks.

STMIA and STMEA are synonyms for STM. STMEA refers to its use for pushing data onto Empty Ascending stacks.

STMFD is a synonym for STMDB, and refers to its use for pushing data onto Full Descending stacks.

Operation

LDM instructions load the registers in *reglist* with word values from memory addresses based on *Rn*.

STM instructions store the word values in the registers in *reglist* to memory addresses based on *Rn*.

For LDM, LDMIA, LDMFD, STM, STMIA, and STMEA the memory addresses used for the accesses are at 4-byte intervals ranging from *Rn* to *Rn* + 4 * (*n*-1), where *n* is the number of registers in *reglist*. The accesses happens in order of increasing register numbers, with the lowest numbered register using the lowest memory address and the highest number register using the highest memory address. If the write-back suffix is specified, the value of *Rn* + 4 * (*n*-1) is written back to *Rn*.

For LDMDB, LDMEA, STMDB, and STMFD the memory addresses used for the accesses are at 4-byte intervals ranging from *Rn* to *Rn* - 4 * (*n*-1), where *n* is the number of registers in *reglist*. The accesses happen in order of decreasing register numbers, with the highest numbered register using the highest memory address and the lowest number register using the lowest memory address. If the write-back suffix is specified, the value of *Rn* - 4 * (*n*-1) is written back to *Rn*.

The PUSH and POP instructions can be expressed in this form.

Restrictions

In these instructions:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- *Rn* must not be PC.
- *reglist* must not contain SP.
- In any `STM` instruction, *reglist* must not contain PC.
- In any `LDM` instruction, *reglist* must not contain PC if it contains LR.
- *reglist* must not contain *Rn* if you specify the write-back suffix.

When PC is in *reglist* in an `LDM` instruction:

- Bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Example 3-73 Examples

```
LDM     R8,{R0,R2,R9}      ; LDMIA is a synonym for LDM.
STMDB   R1!,{R3-R6,R11,R12}
```

Incorrect examples

```
STM     R5!,{R5,R4,R9} ; Value stored for R5 is unpredictable.
LDM     R2, {}         ; There must be at least one register in the list.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.8 PLD

Preload Data.

Syntax

```
PLD{cond} [Rn {, #imm}] ; Immediate
```

```
PLD{cond} [Rn, Rm {, LSL #shift}] ; Register
```

```
PLD{cond} label ; Literal
```

Where:

cond	Is an optional condition code.
Rn	Is the base register.
imm	Is the + or - immediate offset used to form the address. This offset can be omitted, meaning an offset of 0.
Rm	Is the optionally shifted offset register.
shift	Specifies the shift to apply to the value read from *<Rm>*, in the range 0-3. If this option is omitted, a shift by 0 is assumed.
label	The label of the literal item that is likely to be accessed in the near future.

Operation

`PLD` signals the memory system that data memory accesses from a specified address are likely in the near future. If the address is cacheable then the memory system responds by pre-loading the cache line containing the specified address into the data cache. If the address is not cacheable, or the data cache is disabled, this instruction behaves as no operation.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.9 PUSH and POP

Push registers onto, and pop registers off a full-descending stack.

Syntax

`PUSH{cond} reglist`

`POP{cond} reglist`

Where:

cond	Is an optional condition code.
reglist	Is a non-empty list of registers, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

`PUSH` and `POP` are synonyms for `STMDB` and `LDM` (or `LDMIA`) with the memory addresses for the access based on SP, and with the final address for the access written back to the SP. `PUSH` and `POP` are the preferred mnemonics in these cases.

Operation

`PUSH` stores registers on the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

`POP` loads registers from the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address.

`PUSH` uses the value in the SP register minus four as the highest memory address, `POP` uses the value in the SP register as the lowest memory address, implementing a full-descending stack. On completion, `PUSH` updates the SP register to point to the location of the lowest store value, `POP` updates the SP register to point to the location above the highest location loaded.

If a `POP` instruction includes PC in its reglist, a branch to this location is performed when the POP instruction has completed. Bit[0] of the value read for the PC is used to update the APSR T-bit. This bit must be 1 to ensure correct operation.

Restrictions

In these instructions:

- *reglist* must not contain SP.
- For the `PUSH` instruction, *reglist* must not contain PC.
- For the `POP` instruction, *reglist* must not contain PC if it contains LR.

When PC is in *reglist* in a `POP` instruction:

- Bit[0] of the value loaded to the PC must be 1 for correct execution, and a branch occurs to this halfword-aligned address.
- If the instruction is conditional, it must be the last instruction in the IT block.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-74 Examples

```
PUSH {R0,R4-R7} ; Push R0,R4,R5,R6,R7 onto the stack
PUSH {R2,LR}    ; Push R2 and the link-register onto the stack
POP  {R0,R6,PC} ; Pop r0,r6 and PC from the stack, then branch to the new PC.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.10 LDA and STL

Load-Acquire and Store-Release.

Syntax

`op{type}{cond} Rt, [Rn]`

Where:

op	Is one of: **LDA** Load-Acquire Register. **STL** Store-Release Register.
type	Is one of: **B** Unsigned byte, zero extend to 32 bits on loads. **H** Unsigned halfword, zero extend to 32 bits on loads..
cond	Is an optional condition code.
Rt	Is the register to load or store.
Rn	Is the register on which the memory address is based.

Operation

`LDA`, `LDAB`, and `LDAH` loads word, byte, and halfword data respectively from a memory address. If any loads or stores appear after a load-acquire in program order, then all observers are guaranteed to observe the load-acquire before observing the loads and stores. Loads and stores appearing before a load-acquire are unaffected.

`STL`, `STLB`, and `STLH` stores word, byte, and halfword data respectively to a memory address. If any loads or stores appear before a store-release in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after a store-release are unaffected.

In addition, if a store-release is followed by a load-acquire, each observer is guaranteed to observe them in program order.

There is no requirement that a load-acquire and store-release be paired.

All store-release operations are multi-copy atomic, meaning that in a multiprocessing system, if one observer observes a write to memory because of a store-release operation, then all observers observe it. Also, all observers observe all such writes to the same location in the same order.

Restrictions

The address specified must be naturally aligned, or an alignment fault is generated.

The PC must not use SP for *Rt*.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.11 LDREX and STREX

Load and Store Register Exclusive.

Syntax

LDREX{*cond*} *Rt*, [*Rn* {, #*offset*}]

STREX{*cond*} *Rd*, *Rt*, [*Rn* {, #*offset*}]

LDREXB{*cond*} *Rt*, [*Rn*]

STREXB{*cond*} *Rd*, *Rt*, [*Rn*]

LDREXH{*cond*} *Rt*, [*Rn*]

STREXH{*cond*} *Rd*, *Rt*, [*Rn*]

Where:

cond	Is an optional condition code.
Rd	Is the destination register for the returned status.
Rt	Is the register to load or store.
Rn	Is the register on which the memory address is based.
offset	Is an optional offset applied to the value in *Rn*. If *offset* is omitted, the address is the value in *Rn*.

Operation

LDREX, LDREXB, and LDREXH load a word, byte, and halfword respectively from a memory address.

STREX, STREXB, and STREXH attempt to store a word, byte, and halfword respectively to a memory address. The address used in any Store-Exclusive instruction must be the same as the address in the most recently executed Load-exclusive instruction. The value stored by the Store-Exclusive instruction must also have the same data size as the value loaded by the preceding Load-exclusive instruction. This means software must always use a Load-exclusive instruction and a matching Store-Exclusive instruction to perform a synchronization operation.

If a Store-Exclusive instruction performs the store, it writes 0 to its destination register. If it does not perform the store, it writes 1 to its destination register. If the Store-Exclusive instruction writes 0 to the destination register, it is guaranteed that no other process in the system has accessed the memory location between the Load-exclusive and Store-Exclusive instructions.

For reasons of performance, keep the number of instructions between corresponding Load-Exclusive and Store-Exclusive instruction to a minimum.

Note

The result of executing a Store-Exclusive instruction to an address that is different from that used in the preceding Load-Exclusive instruction is unpredictable.

Restrictions

In these instructions:
- Do not use PC.
- Do not use SP for *Rd* and *Rt*.
- For STREX, *Rd* must be different from both *Rt* and *Rn*.
- The value of *offset* must be a multiple of four in the range 0-1020.

Condition flags

These instructions do not change the flags.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Example 3-75 Examples

```
    MOV     R1, #0x1            ; Initialize the ‘lock taken’ value
try
    LDREX   R0, [LockAddr]      ; Load the lock value
    CMP     R0, #0              ; Is the lock free?
    ITT     EQ                  ; IT instruction for STREXEQ and CMPEQ
    STREXEQ R0, R1, [LockAddr]  ; Try and claim the lock
    CMPEQ   R0, #0              ; Did this succeed?
    BNE     try                 ; No – try again
    ....                        ; Yes – we have the lock.
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.12 LDAEX and STLEX

Load-Acquire and Store Release Exclusive.

Syntax

op{*type*} *Rt*, [*Rn*]

Where:

op Is one of:

LDAEX

Load Register.

STLEX

Store Register.

type Is one of:

B	Unsigned byte, zero extend to 32 bits on loads.
H	Unsigned halfword, zero extend to 32 bits on loads..

cond is an optional condition code.

Rd is the destination register for the returned status.

Rt is the register to load or store.

Rn is the register on which the memory address is based.

Operation

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 Shared Memory attribute, marks the physical address as exclusive access for the executing core in a global monitor.
- Causes the core that executes to indicate an active exclusive access in the local monitor.
- If any loads or stores appear after `LDAEX` in program order, then all observers are guaranteed to observe the `LDAEX` before observing the loads and stores. Loads and stores appearing before `LDAEX` are unaffected.

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 core has exclusive access to the memory addressed:

- *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.

- If any loads or stores appear before `STLEX` in program order, then all observers are guaranteed to observe the loads and stores before observing the store-release. Loads and stores appearing after `STLEX` are unaffected.

Note

All store-release operations are multi-copy atomic.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Restrictions

In these instructions:

- Do not use PC.
- Do not use SP for *Rd* and *Rt*.
- For `STLEX`, *Rd* must be different from both *Rt* and *Rn*.

Condition flags

These instructions do not change the flags.

Example 3-76 Examples

```
lock
   MOV R1, #0x1             ; Initialize the ‘lock taken’ value try
   LDAEX R0, [LockAddr]     ; Load the lock value
   CMP R0, #0               ; Is the lock free?
   BNE try                  ; No – try again
   STREX R0, R1, [LockAddr] ; Try and claim the lock
   CMP R0, #0               ; Did this succeed?
   BNE try                  ; No – try again
                            ; Yes – we have the lock.
unlock

   MOV r1, #0
   STL r1, [r0]
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

3.32.13 CLREX

Clear Exclusive.

Syntax

`CLREX{cond}`

Where:

cond Is an optional condition code.

Operation

Use `CLREX` to make the next `STREX`, `STREXB`, or `STREXH` instruction write 1 to its destination register and fail to perform the store. `CLREX` enables compatibility with other Arm Cortex processors that have to force the failure of the store exclusive if the exception occurs between a load-exclusive instruction and the matching store-exclusive instruction in a synchronization operation. In Cortex-M processors, the local exclusive access monitor clears automatically on an exception boundary, so exception handlers using `CLREX` are optional.

Condition flags

This instruction does not change the flags.

Example 3-77 Examples

```
CLREX
```

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 4

Cortex®-M55 Processor-level components and system registers , Reference Material

This chapter presents the reference material for the Arm Cortex-M55 processor-level components and system register descriptions in a User Guide.

It contains the following sections:

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.1 The Cortex®-M55 system registers

The Cortex-M55 system registers are a combination of system control, implementation control, IMPLEMENTATION DEFINED memory system, and power mode control registers. These registers also include architecturally defined registers that are associated with interrupt, event monitoring, memory authentication, and floating-point and vector functionality.

In register descriptions:

- The register type is described as follows:

  RW Read and write.

  RO Read-only.

  WO Write-only.

  RAZ Read As Zero.

  WI Write Ignored.

- The required privilege gives the privilege level that is required to access the register, as follows:

  Privileged Only privileged software can access the register.

  Unprivileged Both unprivileged and privileged software can access the register.

- In an implementation with the Security Extension, the peripheral registers are banked in Secure and Non-secure state. The Non-secure registers can be accessed in Secure state by using an aliased address at offset `0x00020000` from the normal register address. The alias locations are always RAZ/WI if accessed from Non-secure state.

Note

Attempting to access a privileged register from unprivileged software results in a BusFault.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.2 Nested Vectored Interrupt Controller

This section describes the *Nested Vectored Interrupt Controller* (NVIC) and the registers it uses.

The NVIC supports:

- 1-480 interrupts.
- A programmable priority value of 0-255. A higher value corresponds to a lower priority, so value 0 is the highest interrupt priority. In an implementation with the Security Extension, in Non-secure state, the priority also depends on the value of AIRCR.PRIS.
- Level and pulse detection of interrupt signals.
- Interrupt tail-chaining.
- An external *Non-Maskable Interrupt* (NMI).
- An optional *Internal Wake-up Interrupt Controller* (IWIC) and *External Wake-up Interrupt Controller* (EWIC) interface.
- Late arriving interrupts.

The processor automatically stacks its state on exception entry and unstacks this state on exception exit, with no instruction overhead. This provides low latency exception handling.

The following table shows the hardware implementation of NVIC registers. In an implementation with the Security Extension, register fields that are associated with interrupts designated as Secure in the ITNS register are always RAZ/WI if accessed from Non-secure state.

Table 4-1 NVIC registers summary

Address	Name	Type	Required privilege	Reset value	Description
`0xE000E100-0xE000E13C`	NVIC_ISER0-NVIC_ISER15	RW	Privileged	`0x00000000`	*4.2.2 Interrupt Set Enable Registers* on page 4-467
`0xE000E180-0xE000E1BC`	NVIC_ICER0-NVIC_ICER15	RW	Privileged	`0x00000000`	*4.2.3 Interrupt Clear Enable Registers* on page 4-468
`0xE000E200-0xE000E23C`	NVIC_ISPR0-NVIC_ISPR15	RW	Privileged	`0x00000000`	*4.2.4 Interrupt Set Pending Registers* on page 4-469
`0xE000E280-0xE000E2BC`	NVIC_ICPR0-NVIC_ICPR15	RW	Privileged	`0x00000000`	*4.2.8 Interrupt Clear Pending Registers* on page 4-471
`0xE000E300-0xE000E33C`	NVIC_IABR0-NVIC_IABR15	RO	Privileged	`0x00000000`	*4.2.5 Interrupt Active Bit Registers* on page 4-469
`0xE000E380-0xE000E3BC`	NVIC_ITNS0-NVIC_ITNS15	RW[j]	Privileged	`0x00000000`	*4.2.6 Interrupt Target Non-secure Registers* on page 4-470.
`0xE000E400-0xE000E5DC`	NVIC_IPR0-NVIC_IPR119	RW	Privileged	`0x00000000`	*4.2.7 Interrupt Priority Registers* on page 4-470
`0xE000EF00`	STIR	WO	Configurable[k]	`0x00000000`	*4.2.9 Software Trigger Interrupt Register* on page 4-472

4.2.1 Accessing the NVIC registers using CMSIS

CMSIS functions enable software portability between different Cortex-M profile processors.

j ITNS is RAZ/WI from the Non-Secure state.
k See the register description for more information.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

To access the NVIC registers when using CMSIS, use the following functions:

Table 4-2 CMSIS access NVIC functions

CMSIS function	Description
`void NVIC_SetPriorityGrouping (uint32_t PriorityGroup)`	Set priority grouping
`uint32_t NVIC_GetPriorityGrouping (void)`	Read the priority grouping
`void NVIC_EnableIRQ (IRQn_Type IRQn)`	Enable a device-specific interrupt
`uint32_t NVIC_GetEnableIRQ (IRQn_Type IRQn)`	Get a device-specific interrupt enable status.
`void NVIC_DisableIRQ (IRQn_Type IRQn)`	Disable a device-specific interrupt
`uint32_t NVIC_GetPendingIRQ (IRQn_Type IRQn)`	Get the pending device-specific interrupt
`void NVIC_SetPendingIRQ (IRQn_Type IRQn)`	Set a device-specific interrupt to pending
`void NVIC_ClearPendingIRQ (IRQn_Type IRQn)`	Clear a device-specific interrupt from pending
`uint32_t NVIC_GetActive (IRQn_Type IRQn)`	Get the device-specific interrupt active
`void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)`	Set the priority for an interrupt
`uint32_t NVIC_GetPriority (IRQn_Type IRQn)`	Get the priority of an interrupt
`uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)`	Encodes priority
`void NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t *pPreemptPriority, uint32_t *pSubPriority)`	Decode the interrupt priority
`uint32_t NVIC_GetVector (IRQn_Type IRQn)`	Read interrupt vector
`void NVIC_SetVector (IRQn_Type IRQn, uint32_t vector)`	Modify interrupt vector
`void NVIC_SystemReset (void)`	Reset the system
`uint32_t NVIC_GetTargetState (IRQn_Type IRQn)`	Get interrupt target state
`uint32_t NVIC_SetTargetState (IRQn_Type IRQn`	Set interrupt target state
`uint32_t NVIC_ClearTargetState (IRQn_Type IRQn)`	Clear interrupt target state

Note

The input parameter `IRQn` is the IRQ number. For more information on CMSIS NVIC functions, see *http://arm-software.github.io/CMSIS_5/Core/html/group__NVIC__gr.html*

4.2.2 Interrupt Set Enable Registers

The NVIC_ISER0-NVIC_ISER15 registers enable interrupts, and show which interrupts are enabled.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension:
- The register bits can be RAZ/WI depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

In an implementation with the Security Extension, these registers are not banked between Security states.

The bit assignments are:

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-3 NVIC_ISERn bit assignments

Bits	Name	Function
[31:0]	SETENA	Interrupt set-enable bits. For SETENA[m] in NVIC_ISERn, allows interrupt 32n+m to be accessed. Write: **0** No effect. **1** Enable interrupt 32n+m. Read: **0** Interrupt 32n+m disabled. **1** Interrupt 32n+m enabled.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

4.2.3 Interrupt Clear Enable Registers

The NVIC_ICER0-NVIC_ICER15 registers disable interrupts, and show which interrupts are enabled.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension:
- The register bits can be RAZ/WI from Non-secure state depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

The bit assignments are:

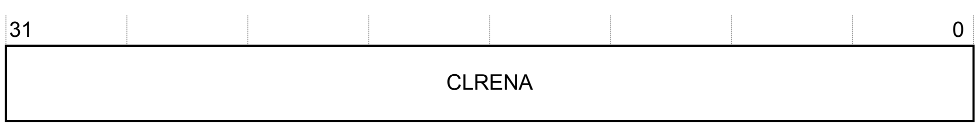


Table 4-4 NVIC_ICERn bit assignments

Bits	Name	Function
[31:0]	CLRENA	Interrupt clear-enable bits. For SETENA[m] in NVIC_ICERn, allows interrupt 32n + m to be accessed. Write: **0** No effect. **1** Disable interrupt 32n+m. Read: **0** Interrupt 32n+m disabled. **1** Interrupt 32n+m enabled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.2.4 Interrupt Set Pending Registers

The NVIC_ISPR0-NVIC_ISPR15 registers force interrupts into the pending state, and shows which interrupts are pending.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension:
- The register bits can be RAZ/WI from Non-secure state depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

The bit assignments are:

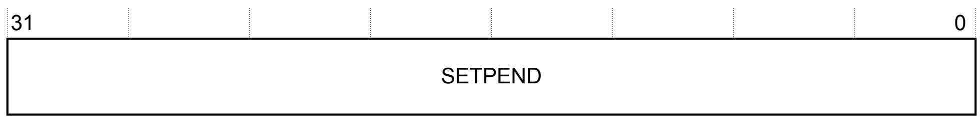


Table 4-5 NVIC_ISPRn bit assignments

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits. For SETPEND[m] in NVIC_ISPRn, allows interrupt 32n + m to be accessed. Write: **0** No effect. **1** Pend interrupt 32n + m. Read: **0** Interrupt 32n + m is not pending. **1** Interrupt 32n + m pending.

Note

Writing 1 to the NVIC_ISPR bit corresponding to:
- An interrupt that is pending has no effect.
- A disabled interrupt sets the state of that interrupt to pending.

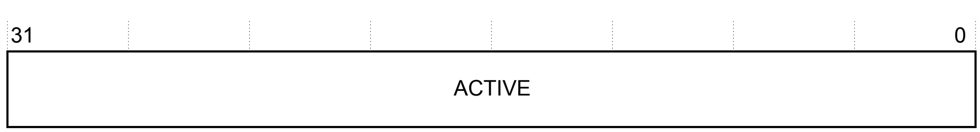
4.2.5 Interrupt Active Bit Registers

The NVIC_IABR0-NVIC_IABR15 registers indicate the active state of each interrupt.

This register is a 32-bit RO register.

In an implementation with the Security Extension:
- The register bits can be RAZ/WI from Non-secure state depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

The bit assignments are:



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-6 NVIC_IABRn bit assignments

Bits	Name	Function
[31:0]	ACTIVE	Active state bits. For ACTIVE[m] in NVIC_IABRn, indicates the active state for interrupt 32n+m. **0** The interrupt is not active. **1** The interrupt is active.

4.2.6 Interrupt Target Non-secure Registers

In an implementation with the Security Extension, the NVIC_ITNS0-NVIC_ITNS15 registers determine, for each group of 32 interrupts, whether each interrupt targets Non-secure or Secure state. Otherwise, This register is RAZ/WI.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension, this register is accessible from Secure state only.

The bit assignments are:

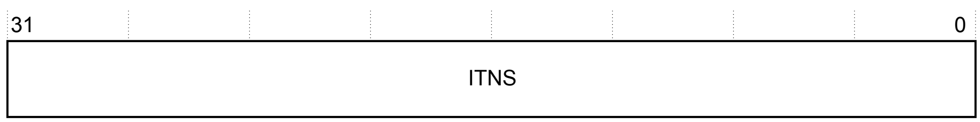


Table 4-7 NVIC_ITNSn bit assignments

Bits	Name	Function
[31:0]	ITNS	Interrupt Targets Non-secure bits. For ITNS[m] in NVIC_ITNSn, this field indicates and allows modification of the target Security state for interrupt 32n+m. **0** The interrupt targets Secure state. **1** The interrupt targets Non-secure state.

4.2.7 Interrupt Priority Registers

The NVIC_IPR0-NVIC_IPR119 registers provide a priority field for each interrupt. These registers are word, halfword, and byte accessible.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for their attributes.

Each register holds four priority fields as shown:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

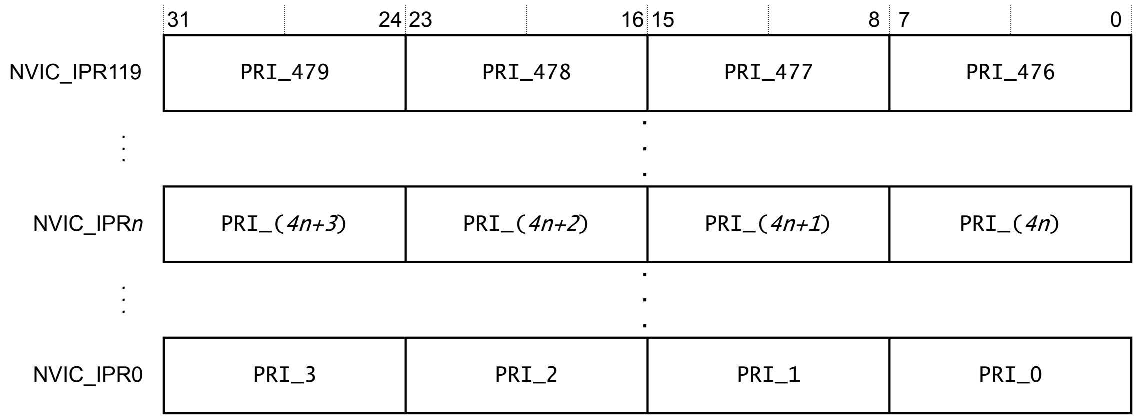


Table 4-8 NVIC_IPRn bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value. The priority depends on the value of PRIS for exceptions targeting the Non-secure state. The processor implements n MSBs in each field. If n<8, the 8-n LSBs are RES0.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

See *4.2.1 Accessing the NVIC registers using CMSIS* on page 4-466 for more information about the access to the interrupt priority array, which provides the software view of the interrupt priorities.

Find the NVIC_IPR number and byte offset for interrupt *M* as follows:

- The corresponding NVIC_IPR number, *N*, is given by *N* = *N* DIV 4.
- The byte offset of the required Priority field in this register is *M* MOD 4, where:
  — Byte offset 0 refers to register bits[7:0].
  — Byte offset 1 refers to register bits[15:8].
  — Byte offset 2 refers to register bits[23:16].
  — Byte offset 3 refers to register bits[31:24].

In an implementation with the Security Extension:

- Priority values depend on the value of PRIS.
- The register bits can be RAZ/WI depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

4.2.8 Interrupt Clear Pending Registers

The NVIC_ICPR0-NVIC_ICPR15 registers remove the pending state from interrupts, and shows which interrupts are pending.

See the register summary in *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension:

- The register bits can be RAZ/WI depending on the value of NVIC_ITNS.
- These registers are not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The bit assignments are:

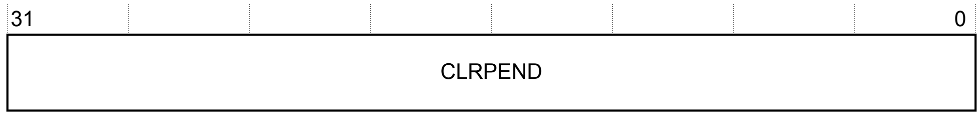


Table 4-9 NVIC_ICPRn bit assignments

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits. Write: **0** No effect. **1** Clear pending state of interrupt 32n+m. Read: **0** Interrupt 32n+m is not pending. **1** Interrupt 32n+m is pending.

Note

Writing 1 to an NVIC_ICPR bit does not affect the active state of the corresponding interrupt.

4.2.9 Software Trigger Interrupt Register

Write to the STIR to generate an interrupt from software.

When the USERSETMPEND bit in the CCR is set to 1, unprivileged software can access the STIR.

Note

Only privileged software can enable unprivileged access to the STIR. This register returns 0 on a read transaction.

See *4.2 Nested Vectored Interrupt Controller* on page 4-466 for the register attributes.

In an implementation with the Security Extension, this register is not banked between Security states.

The bit assignments are:

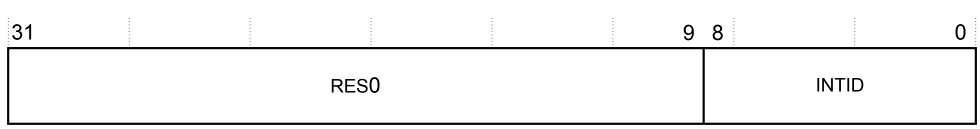


Table 4-10 STIR bit assignments

Bits	Field	Function
[31:9]	-	Reserved, RES0.
[8:0]	INTID	Interrupt ID of the interrupt to trigger, in the range 0-479. For example, a value of `0x03` specifies interrupt IRQ3.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.2.10 Level-sensitive and pulse interrupts

The processor supports both level-sensitive and pulse interrupts. Pulse interrupts are also described as edge-triggered interrupts.

A level-sensitive interrupt is held asserted until the peripheral deasserts the interrupt signal. Typically this happens because the ISR accesses the peripheral, causing it to clear the interrupt request. A pulse interrupt is an interrupt signal sampled synchronously on the rising edge of the processor clock. To ensure that the NVIC detects the interrupt, the peripheral must assert the interrupt signal for at least one clock cycle, during which the NVIC detects the pulse and latches the interrupt.

When the processor enters the ISR, it automatically removes the pending state from the interrupt.

For a level-sensitive interrupt, if the signal is not deasserted before the processor returns from the ISR, the interrupt becomes pending again, and the processor must execute its ISR again. This means that the peripheral can hold the interrupt signal asserted until it no longer requires servicing.

Hardware and software control of interrupts

The processor latches all interrupts. A peripheral interrupt becomes pending for one of the following reasons:

- The NVIC detects that the interrupt signal is active and the corresponding interrupt is not active.
- The NVIC detects a rising edge on the interrupt signal.
- Software writes to the corresponding ISPR bit.

A pending interrupt remains pending until one of the following occurs:

- The processor enters the ISR for the interrupt. This changes the state of the interrupt from pending to active. Then:
  - For a level-sensitive interrupt, when the processor returns from the ISR, the NVIC samples the interrupt signal. If the signal is asserted, the state of the interrupt changes to pending, which might cause the processor to immediately reenter the ISR. Otherwise, the state of the interrupt changes to inactive.
  - For a pulse interrupt, the NVIC continues to monitor the interrupt signal, and if this is pulsed the state of the interrupt changes to pending and active. In this case, when the processor returns from the ISR the state of the interrupt changes to pending, which might cause the processor to immediately reenter the ISR.

    If the interrupt signal is not pulsed while the processor is in the ISR, when the processor returns from the ISR the state of the interrupt changes to inactive.
- Software writes to the corresponding ICPR bit.

  For a level-sensitive interrupt, if the interrupt signal is still asserted, the state of the interrupt does not change. Otherwise, the state of the interrupt changes to inactive.

  For a pulse interrupt, state of the interrupt changes to:
  - Inactive, if the state was pending.
  - Active, if the state was active and pending.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3 System Control and Implementation Control Block

The *System Control Block* (SCB) and *Implementation Control Block* (ICB) provide system implementation information and system control that includes configuration, control, and reporting of system exceptions.

4.3.1 System control block registers summary

The following table shows the SCB registers that provide system control and configuration information.

Table 4-11 SCB register summary

Address	Name	Type	Reset value	Description
`0xE000ECFC`	REVIDR	RO	`0x00000000`	Revision ID register
`0xE000ED00`	CPUID	RO	`0x410FD221`	*4.3.10 CPUID Base Register* on page 4-487
`0xE000ED04`	ICSR	RW	`0x00000000`	*4.3.18 Interrupt Control and State Register* on page 4-504
`0xE000ED08`	VTOR	RW	UNKNOWN **Note** • Bits [31:7] of VTOR_S are based on **INITSVTOR[31:7]**. The Secure version of this register does not exist if the Security extension is not configured and only INITNSVTOR[31:7] exists. Bits [6:0] are RES0. • Bits [31:7] of VTOR_NS are based on INITNSVTOR[31:7].	*4.3.35 Vector Table Offset Register* on page 4-529
`0xE000ED0C`	AIRCR	RW	UNKNOWN **Note** Bit [15] of this register depends on input signal **CFGBIGEND**. The other bits reset to zero.	*4.3.4 Application Interrupt and Reset Control Register* on page 4-478
`0xE000ED10`	SCR	RW	`0x00000000`	*4.3.31 System Control Register* on page 4-518
`0xE000ED14`	CCR	RW	`0x00000201`	*4.3.11 Configuration and Control Register* on page 4-488
`0xE000ED18`	SHPR1	RW	`0x00000000`	*System Handler Priority Register 1* on page 4-525
`0xE000ED1C`	SHPR2	RW	`0x00000000`	*System Handler Priority Register 2* on page 4-525
`0xE000ED20`	SHPR3	RW	`0x00000000`	*System Handler Priority Register 3* on page 4-526
`0xE000ED24`	SHCSR	RW	`0x00000000`	*4.3.32 System Handler Control and State Register* on page 4-521

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-11 SCB register summary (continued)

Address	Name	Type	Reset value	Description
0xE000ED28	CFSR	RW	0x00000000	*4.3.12 Configurable Fault Status Register* on page 4-491
0xE000ED28	MMFSR	RW	0x00	*MemManage Fault Status Register* on page 4-496
0xE000ED29	BFSR	RW	0x00	*BusFault Status Register* on page 4-494
0xE000ED2A	UFSR	RW	0x0000	*UsageFault Status Register* on page 4-492
0xE000ED2C	HFSR	RW	0x00000000	*4.3.17 HardFault Status Register* on page 4-503
0xE000ED34	MMFAR	RW	UNKNOWN	*4.3.25 MemManage Fault Address Register* on page 4-513
0xE000ED38	BFAR	RW	UNKNOWN	*4.3.5 Bus Fault Address Register* on page 4-482
0xE000ED3C	AFSR	RAZ/WI	0x00000000	*4.3.2 Auxiliary Fault Status Register* on page 4-476
0xE000ED40	ID_PFR0	RO	0x20000030	*4.3.14 Processor Feature Register 0* on page 4-499
0xE000ED44	ID_PFR1	RO	0x000002X0	*4.3.15 Processor Feature Register 1* on page 4-500
0xE000ED48	ID_DFR0	RO	0x10X00000	
0xE000ED4C	ID_AFR0	RO	0x00000000	*4.3.3 Auxiliary Feature Register 0* on page 4-478
0xE000ED50	ID_MMFR0	RO	0x00111040	*4.3.26 Memory Model Feature Register 0* on page 4-514
0xE000ED54	ID_MMFR1	RO	0x00000000	*4.3.27 Memory Model Feature Register 1* on page 4-515
0xE000ED58	ID_MMFR2	RO	0x01000000	*4.3.28 Memory Model Feature Register 2* on page 4-515
0xE000ED5C	ID_MMFR3	RO	0x00000011	*4.3.29 Memory Model Feature Register 3* on page 4-516
0xE000ED60	ID_ISAR0	RO	0x011X3110 **Note** ID_ISAR0[19:16] depend on whether the external coprocessor interface is included in the processor. When the value of this register is 0b0100, this indicates that the coprocessor interface is supported.	*4.3.19 Instruction Set Attribute Register 0* on page 4-507
0xE000ED64	ID_ISAR1	RO	0x02212000	*4.3.20 Instruction Set Attribute Register 1* on page 4-508

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-11 SCB register summary (continued)

Address	Name	Type	Reset value	Description
`0xE000ED68`	ID_ISAR2	RO	`0x20232232`	*4.3.21 Instruction Set Attribute Register 2* on page 4-509
`0xE000ED6C`	ID_ISAR3	RO	`0x01111131`	*4.3.22 Instruction Set Attribute Register 3* on page 4-510
`0xE000ED70`	ID_ISAR4	RO	`0x01310132`	*4.3.23 Instruction Set Attribute Register 4* on page 4-512
`0xE000ED74`	ID_ISAR5	RO	`0x00000000`	*4.3.24 Instruction Set Attribute Register 5* on page 4-513
`0xE000ED78`	CLIDR	RO	`0xXXX0000X` **Note** CLIDR[31:21] and CLIDR[2:0] depend on the cache configuration of the processor.	*4.3.6 Cache Level ID Register* on page 4-483
`0xE000ED7C`	CTR	RO	• If an instruction cache or data cache is included, then the reset value is `0x8303C003`. • If an instruction cache or data cache is not included, then the reset value is `0x00000000`.	*4.3.8 Cache Type Register* on page 4-485
`0xE000ED80`	CSSIDR	RO	`0xXXXXXXXX` **Note** CCSIDR depends on the CSSELR setting and L1 cache configuration.	*4.3.9 Current Cache Size ID Register* on page 4-486
`0xE000ED84`	CSSELR	RW	`0x00000000`	*4.3.7 Cache Size Selection Register* on page 4-484
`0xE000ED88`	CPACR	RW	`0x00000000`	*4.3.13 Coprocessor Access Control Register* on page 4-498
`0xE000ED8C`	NSACR	RW	UNKNOWN	*4.3.30 Non-secure Access Control Register* on page 4-517

4.3.2 Auxiliary Fault Status Register

The AFSR provides fault status information.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.
The register is set to zero at reset and all fields are cleared to zero by a software write of any value. AFSR bits [31:21] are only valid if BFSR.IBUSERR is set. AFSR bits [20:10] are only valid is BFSR.PRECISEERR is set.
AFSR bits [9:0] are only valid if BFSR.IMPRECISEERR is set.
If multiple faults occur, the AFSR indicates the types of all the faults that have occurred. For more information on BFSR, see the *Arm®v8-M Architecture Reference Manual*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This register is always implemented.

Attributes

A 32-bit RO register located at `0xE000ED3C`.
Non-secure alias is provided using AFSR_NS, located at `0xE002ED3C`.
If the Security Extension is implemented, non-secure alias is provided using AFSR_NS, located at `0xE002ED3C`.
This register is not banked between Security states.
If the Security Extension is implemented, this register is not banked between Security states.

The following figure shows the AFSR bit assignments.

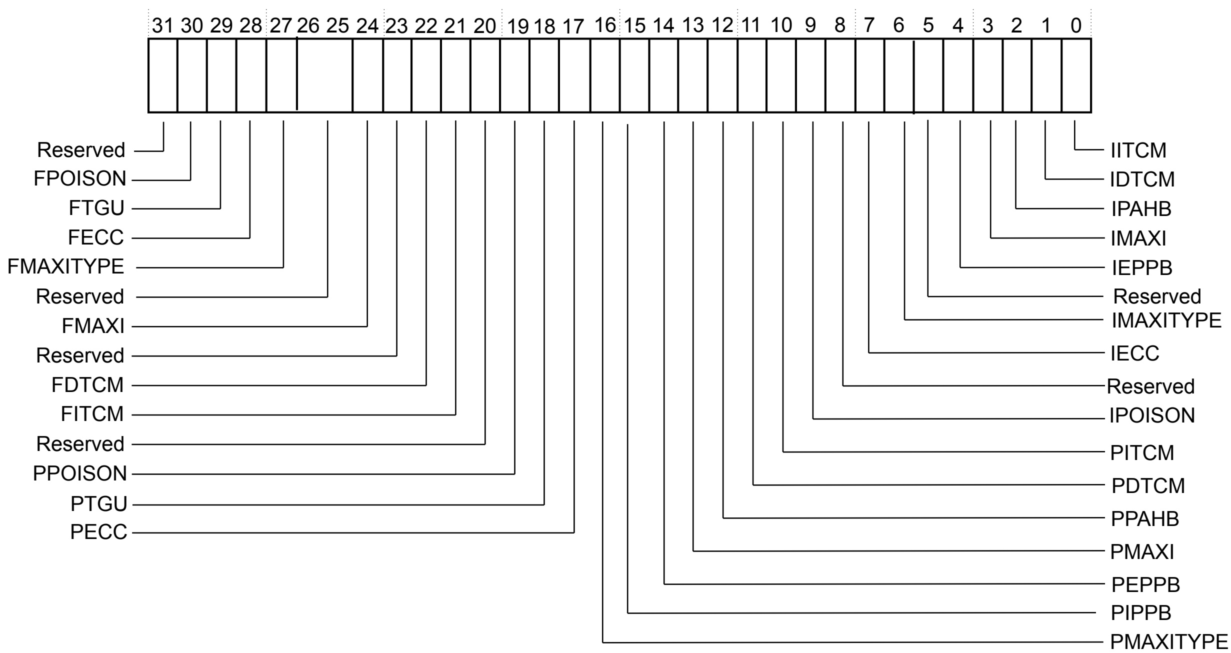


Figure 4-1 AFSR bit assignments

The following table describes the AFSR bit assignments.

Table 4-12 AFSR bit assignments

Bits	Name	Type	Description
[31]	Reserved	-	RES0
[30]	FPOISON	RO	Fetch fault caused by **RPOISON** or TEBRx.POISON.
[29]	FTGU	-	Fetch fault caused by *TCM Gate Unit* (TGU) security violation.
[28]	FECC	RO	Fetch fault caused by uncorrectable *Error Correcting Code* (ECC) error.
[27]	FMAXITYPE	RO	AXI response that caused the fetch fault. Only valid when AFSR.FMAXI is 1. `0b0` SLVERR `0b1` DECERR
[26:25]	Reserved	-	RES0
[24]	FMAXI	RO	Fetch fault on *Master AXI* (M-AXI) interface.
[23]	Reserved	-	RES0

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-12 AFSR bit assignments (continued)

Bits	Name	Type	Description
[22]	FDTCM	RO	Fetch fault on *Data Tightly Coupled Memory* (DTCM) interface.
[21]	FITCM	RO	Fetch fault on *Instruction Tightly Coupled Memory* (ITCM) interface.
[20]	Reserved	-	RES0
[19]	PPOISON	RO	Precise fault caused by **RPOISON** or TEBRx.POISON.
[18]	PTGU	RO	Precise fault caused by TGU security violation.
[17]	PECC	RO	Precise fault caused by uncorrectable ECC error.
[16]	PMAXITYPE	RO	AXI response that caused the precise fault. Only valid when AFSR.PMAXI is 1. `0b0` SLVERR `0b1` DECERR
[15]	PIPPB	RO	Precise fault on *Internal Private Peripheral Bus* (EPPB) interface.
[14]	PEPPB	RO	Precise fault on *External Private Peripheral Bus* (EPPB) interface.
[13]	PMAXI	RO	Precise fault on M-AXI interface.
[12]	PPAHB	RO	Precise fault on *Peripheral AHB* (P-AHB) interface.
[11]	PDTCM	RO	Precise fault on DTCM interface.
[10]	PITCM	RO	Precise fault on ITCM interface.
[9]	IPOISON	RO	Imprecise BusFault because of **RPOISON**.
[8]	Reserved	-	RES0
[7]	IECC	RO	Imprecise fault caused by uncorrectable ECC error.
[6]	IMAXITYPE	RO	AXI response that caused the imprecise fault. Only valid when AFSR.IMAXI is 1. `0b0` SLVERR `0b1` DECERR
[5]	Reserved	-	RES0
[4]	IEPPB	RO	Imprecise fault on EPPB interface.
[3]	IMAXI	RO	Imprecise fault on M-AXI interface.
[2]	IPAHB	RO	Imprecise fault on P-AHB interface.
[1]	IDTCM	RO	Imprecise fault on DTCM interface.
[0]	IITCM	RO	Imprecise fault on ITCM interface.

4.3.3 Auxiliary Feature Register 0

The ID_AFR0 is fully Reserved, RES0.

4.3.4 Application Interrupt and Reset Control Register

The AIRCR provides sets or returns interrupt control and reset configuration.

Usage constraints

See *4.3.4 Application Interrupt and Reset Control Register* on page 4-478 for the AIRCR attributes. To write to this register, you must write `0x5FA` to the VECTKEY field, otherwise the processor ignores the write.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This register is always implemented.

Attributes

This register is banked between Security states.
A 32-bit RW register located at `0xE000ED0C`.
Non-secure alias is provided using AIRCR_NS, located at `0xE002ED0C`.

In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the AIRCR bit assignments.

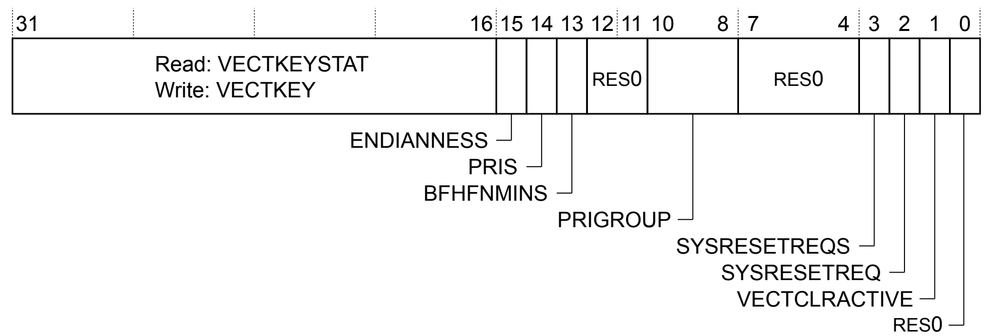


Figure 4-2 AIRCR bit assignments

The following table describes the AIRCR bit assignments.

Table 4-13 AIRCR bit assignments without the Security Extension

Bits	Name	Type	Function
[31:16]	Read: VECTKEYSTAT Write: VECTKEY	RW	Register key: Reads as `0x0FA05`. On writes, write `0x5FA` to VECTKEY, otherwise the write is ignored.
[15]	ENDIANNESS	RO	Data endianness bit: **0** Little-endian. **1** Big-endian.
[14]	PRIS	RAZ/WI	-
[13]	BFHFNMINS	RAO/WI	-
[12:11]	-	-	Reserved, RES0.
[10:8]	PRIGROUP	RW	Interrupt priority grouping field. This field determines the split of group priority from subpriority, see *Binary point* on page 4-481.
[7:4]	-	-	Reserved, RES0.
[3]	SYSRESETREQS	RAZ/WI	-

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-13 AIRCR bit assignments without the Security Extension (continued)

Bits	Name	Type	Function
[2]	SYSRESETREQ	RW	System reset request. This bit allows software or a debugger to request a system reset: **0** Do not request a system reset. **1** Request a system reset. This bit is not banked between Security states.
[1]	VECTCLRACTIVE	WO	Reserved for Debug use. This bit reads as 0. When writing to the register you must write 0 to this bit, otherwise behavior is UNPREDICTABLE.
[0]	-	-	Reserved, RES0.

Table 4-14 AIRCR bit assignments with the Security Extension

Bits	Name	Type	Function
[31:16]	Read: VECTKEYSTAT Write: VECTKEY	RW	Register key: Reads as `0x0FA05`. On writes, write `0x5FA` to VECTKEY, otherwise the write is ignored. This Field is not banked between Security states.
[15]	ENDIANNESS	RO	Data endianness bit: **0** Little-endian. **1** Big-endian. This bit is not banked between Security states.
[14]	PRIS	RW from Secure state and RAZ/WI from Non-secure state. RAZ/WI	Prioritize Secure exceptions. The value of this bit defines whether Secure exception priority boosting is enabled. 0 Priority ranges of Secure and Non-secure exceptions are identical. 1 Non-secure exceptions are de-prioritized. This bit is not banked between Security states. -
[13]	BFHFNMINS	RW from Secure-state and RO from Non-secure state. RAO/WI	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. The possible values are: **0** BusFault, HardFault, and NMI are Secure. **1** BusFault and NMI are Non-secure and exceptions can target Non-secure HardFault. This bit resets to 0. This bit is not banked between Security states. -
[12:11]	-	-	Reserved, RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-14 AIRCR bit assignments with the Security Extension (continued)

Bits	Name	Type	Function
[10:8]	PRIGROUP	RW	Interrupt priority grouping field. This field determines the split of group priority from subpriority, see *Binary point* on page 4-481. This bit is banked between Security states.
[7:4]	-	-	Reserved, RES0.
[3]	SYSRESETREQS	RW from Secure State and RAZ/WI from Non-secure state. RAZ/WI	System reset request, Secure state only. The value of this bit defines whether the SYSRESETREQ bit is functional for Non-secure use: 0 SYSRESETREQ functionality is available to both Security states. 1 SYSRESETREQ functionality is only available to Secure state. This bit resets to zero on a Warm reset. This bit is not banked between Security states. -
[2]	SYSRESETREQ	RW if SYSRESETREQS is 0. When SYSRESETREQS is set to 1, from Non-secure state this bit acts as RAZ/WI. RW	System reset request. This bit allows software or a debugger to request a system reset: **0** Do not request a system reset. **1** Request a system reset. This bit is not banked between Security states.
[1]	VECTCLRACTIVE	WO	Reserved for Debug use. This bit reads as 0. When writing to the register you must write 0 to this bit, otherwise behavior is UNPREDICTABLE. This bit is not banked between Security states.
[0]	-	-	Reserved, RES0.

Note

The processor has external signal, **LOCKSVTAIRCR**, that disables writes to AIRCR.PRIS and AIRCR.BFHFNMINS from software or from a debug agent that is connected to the processor.

Binary point

The PRIGROUP field indicates the position of the binary point that splits the PRI_*n* fields in the Interrupt Priority Registers into separate group priority and subpriority fields.

The following table shows how the PRIGROUP value controls this split.

Table 4-15 Priority grouping

	Interrupt priority level value, PRI_*n*[7:0]			Number of	
PRIGROUP	**Binary point**[l]	**Group priority bits**	**Subpriority bits**	**Group priorities**	**Subpriorities**
`0b000`	bxxxxxxx.y	[7:1]	[0]	128	2
`0b001`	bxxxxxx.yy	[7:2]	[1:0]	64	4

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-15 Priority grouping (continued)

	Interrupt priority level value, PRI_*n*[7:0]			Number of	
PRIGROUP	Binary point[1]	Group priority bits	Subpriority bits	Group priorities	Subpriorities
`0b010`	bxxxxx.yyy	[7:3]	[2:0]	32	8
`0b011`	bxxxx.yyyy	[7:4]	[3:0]	16	16
`0b100`	bxxx.yyyyy	[7:5]	[4:0]	8	32
`0b101`	bxx.yyyyyy	[7:6]	[5:0]	4	64
`0b110`	bx.yyyyyyy	[7]	[6:0]	2	128
`0b111`	b.yyyyyyyy	None	[7:0]	1	256

Note

Determining pre-emption of an exception uses only the group priority field.

4.3.5 Bus Fault Address Register

The MMFAR shows the address of the memory location that caused a *Memory Protection Unit* (MPU) fault.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
A 32-bit RW register located at `0xE000ED38`.
Non-secure alias is provided using BFAR_NS, located at `0xE002ED38`.
. If the Security Extension is implemented, non-secure alias is provided using BFAR_NS, located at `0xE002ED38`.
This register is not banked between Security states.
The Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is 0.
If the Security Extension is implemented, this register is not banked between Security states.
If the Security Extension is implemented, the Non-secure version of this register is RAZ/WI if AIRCR.BFHFNMINS is 0.

The following figure shows the BFAR bit assignments.

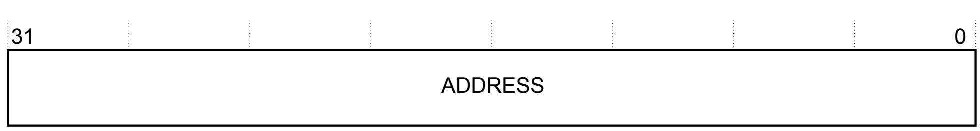


Figure 4-3 BFAR bit assignments

The following table describes the BFAR bit assignments.

1 PRI_*n*[7:0] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-16 BFAR bit assignments

Bits	Name	Type	Description
[31:0]	ADDRESS	RW	Data address for the precise BusFault. This field is valid only when BFSR.BFARVALID is set, otherwise it is UNKNOWN.

4.3.6 Cache Level ID Register

The CLIDR identifies the type of caches implemented and the level of coherency and unification. If an instruction, data cache, or both is not configured in the processor, then CLIDR is `0x00000000`.

Usage constraints
Privileged mode access only. Unprivileged accesses generate a fault.

Configuration

This register is always implemented.

Attributes
32-bit RO register located at `0xE000ED78`.
This register is not banked between Security states.
Non-secure version of this register through CLIDR_NS located at `0xE002ED78`.
If the Security Extension is implemented, the non-secure version of this register through CLIDR_NS located at `0xE002ED78`.

The following figure shows the CLIDR bit assignments.

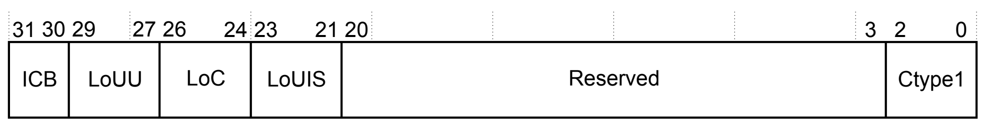


Figure 4-4 CLIDR bit assignments

The following table shows the CLIDR bit assignments.

Table 4-17 CLIDR bit assignments

Bits	Name	Type	Description
[31:30]	ICB	RO	Inner cache boundary. The Cortex-M55 processor supports inner cacheability on the bus. Therefore, this field cannot disclose any information. `0b00` Not disclosed in this mechanism.
[29:27]	LoUU	RO	Level of Unification Uniprocessor. The L1 cache must be cleaned or invalidated when cleaning or invalidating occurs to the point of unification. The options are: `0b000` Caches are not implemented. Therefore, cleaning and invalidation is not required. `0b001` *Level 1* (L1) data cache or instruction cache is implemented. Therefore, cleaning and invalidation are required.
[26:24]	LoC	RO	Level of Coherency. The L1 cache must be cleaned when cleaning occurs to the point of coherency. The options are: `0b000` Caches are not implemented. Therefore, cleaning is not required. `0b001` L1 data cache or instruction cache is implemented. Therefore, cleaning is required.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-17 CLIDR bit assignments (continued)

Bits	Name	Type	Description
[23:21]	LoUIS	RO	Level of Unification Inner Shareable. The L1 cache must be cleaned or invalidated when cleaning or invalidating occurs to the point of unification for the inner shareability domain. The options are: `0b000` Caches are not implemented. Therefore, cleaning and invalidation are not required. `0b001` L1 data cache or instruction cache is implemented. Therefore, cleaning and invalidation are required.
[20:3]	Reserved	-	RES0
[2:0]	Ctype1	RO	*Level 1* (L1) cache type. The options are: `0b000` Caches are not implemented. `0b001` Instruction cache is implemented. `0b010` Data cache is implemented. `0b011` Data cache and instruction cache are implemented.

4.3.7 Cache Size Selection Register

The CSSELR selects the current CCSIDR by specifying the cache level and the type of cache (either instruction or data cache).

Usage constraints
This register is read/write and is accessible in Privileged mode only.

Configurations

This register is always implemented.

Attributes
32-bit RW register located at `0xE000ED84`.
This register is not banked between Security states.
Non-secure version of this register through CSSELR_NS located at `0xE002ED84`.
If the Security Extension is implemented, the non-secure version of this register through CSSELR_NS located at `0xE002ED84`.

This register is banked between Security states. The following figure shows the CSSELR bit assignments.

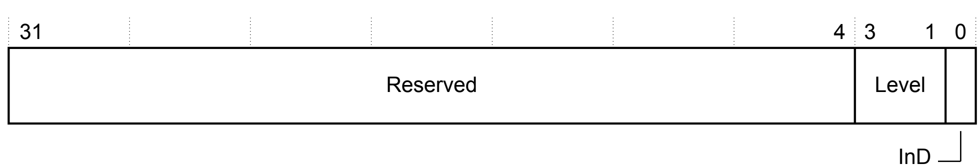


Figure 4-5 CSSELR bit assignments

The following table shows the CSSELR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-18 CSSELR bit assignments

Bits	Name	Type	Function
[31:4]	Reserved	-	RES0
[3:1]	Level	RO	Identifies which cache level to select. `0x0` *Level 1* (L1) cache. This field is RAZ/WI.
[0]	InD	RW	Selects either L1 instruction or data cache. The options are: **0** L1 data cache. **1** L1 instruction cache.

4.3.8 Cache Type Register

The CTR provides information about the architecture of the caches. If the processor is not configured to include caches, then CLIDR is `0x00000000`.

Usage constraints
Privileged mode access only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
32-bit RO register located at `0xE000ED7C`.
This register is not banked between Security states.
Non-secure version of this register through CTR_NS located at `0xE002ED7C`.
If the Security Extension is implemented, the non-secure version of this register through CTR_NS located at `0xE002ED7C`.

The following figure shows the CTR bit assignments.

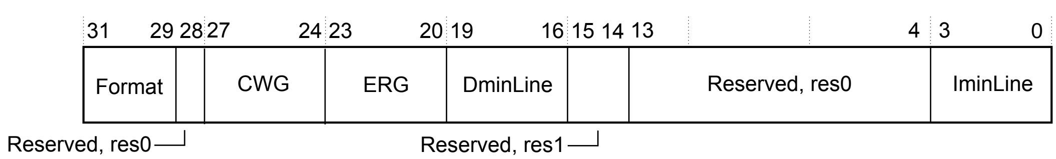


Figure 4-6 CTR bit assignments

The following table shows the CTR bit assignments.

Table 4-19 CTR bit assignments

Bits	Name	Type	Description
[31:29]	Format	RO	Cache type register format. Indicates whether cache type information is provided. `0b100` Cache type information is provided.
[28]	Reserved	-	RES0

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-19 CTR bit assignments (continued)

Bits	Name	Type	Description
[27:24]	CWG	RO	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.
[23:20]	ERG	RO	Exclusives Reservation Granule. Log2 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.
[19:16]	DminLine	RO	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 processor.
[15:14]	Reserved	-	RES1
[13:4]	Reserved	-	RES0
[3:0]	IminLine	RO	Instruction cache minimum line length. Log2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the processor.

4.3.9 Current Cache Size ID Register

The CCSIDR provides information about the architecture of the *Level 1* (L1) instruction or data cache that the CSSELR selects. If the cache corresponding to CSSELR.InD is not included in the processor, then this register reads `0x00000000`.

Usage constraints

Privileged mode access only. Unprivileged accesses generate a fault.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED80`.
This register is not banked between Security states.
Non-secure version of this register through CSSIDR_NS located at `0xE002ED80`.
If the Security Extension is implemented, the non-secure version of this register through CSSIDR_NS located at `0xE002ED80`.

The following figure shows the CCSIDR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

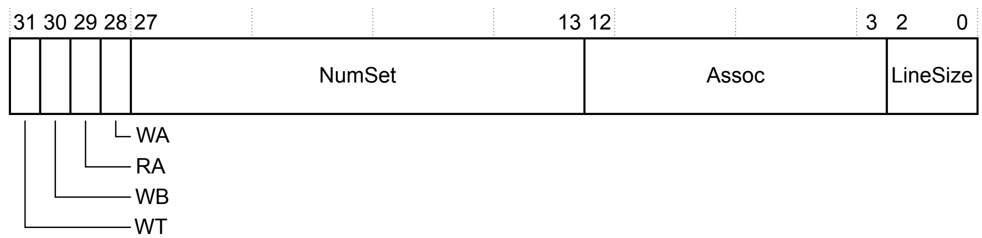


Figure 4-7 CCSIDR bit assignments

The following table shows the CCSIDR bit assignments.

Table 4-20 CCSIDR bit assignments

Bits	Name	Type	Function
[31]	WT	RO	Indicates support available for Write-Through: `0b1` Write-Through support available.
[30]	WB	RO	Indicates support available for Write-Back: `0b1` Write-Back support available.
[29]	RA	RO	Indicates support available for read allocation: `0b1` Read allocation support available.
[28]	WA	RO	Indicates support available for write allocation: `0b1` Write allocation support available.
[27:13]	NumSet	RO	Indicates the number of sets. Cache-size dependent.
[12:3]	Assoc	RO	Indicates associativity. The value depends on the cache that CSSELR selects. When CSSELR.InD=1 (L1 instruction cache): `0x1` 2-way set associative instruction cache. When CSSELR.InD=0 (L1 data cache) `0x3` 4-way set associative data cache.
[2:0]	LineSize	RO	Indicates the number of words in each cache line. `0b1` Represents 32 bytes.

The LineSize field is encoded as 2 less than log(2) of the number of words in the cache line. For example, a value of `0x0` indicates there are four words in a cache line, that is the minimum size for the cache. A value of `0x1` indicates there are eight words in a cache line.

4.3.10 CPUID Base Register

The CPUID Base Register contains the processor part number, version, and implementation information.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CPUID attributes.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED00`.
This register is not banked between Security states.
Non-secure alias is provided using CPUID_NS, located at `0xE002ED00`.

In an implementation with the Security Extension, this register is not banked between Security states.

The following figure shows the CPUID bit assignments.

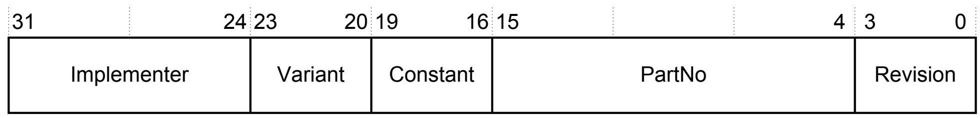


Figure 4-8 CPUID bit assignments

The following table shows the CPUID bit assignments.

Table 4-21 CPUID bit assignments

Bits	Name	Function
[31:24]	Implementer	Implementer code: `0x41` ArmLimited
[23:20]	Variant	Variant number, the n value in the r*n*p*m* product revision identifier: `0x0` Revision 0
[19:16]	Architecture	Reads as `0b1111`, Armv8.1-M with Main Extension
[15:4]	PartNo	Part number of the processor: `0xD22` Cortex-M55
[3:0]	Revision	Revision number, the m value in the r*n*p*m* product revision identifier: `0x1` Patch 1.

4.3.11 Configuration and Control Register

The CCR is a read-only register and indicates some aspects of the behavior of the processor.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CCR attributes.

Configuration

This register is always implemented.

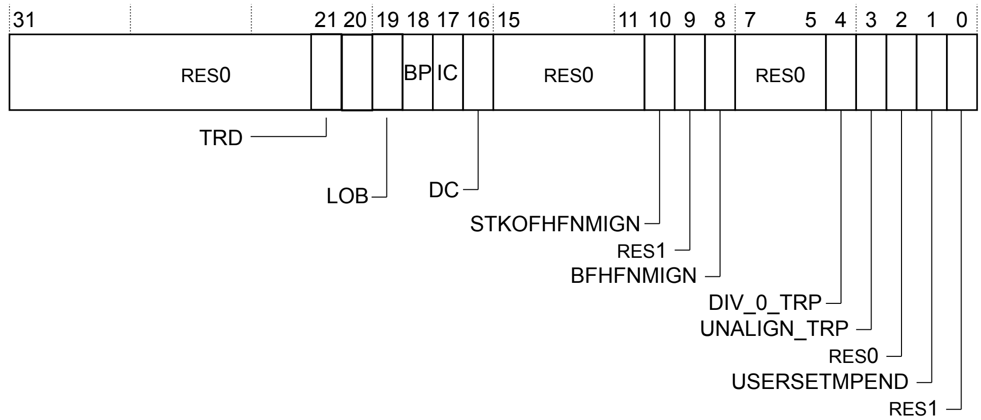
Attributes

A 32-bit RW register located at `0xE000E14`.
Non-secure alias is provided using CCR_NS, located at `0xE002ED14`.
This register is banked between Security states on a bit by bit basis.

In an implementation with the Security Extension, this register is banked between Security states on a bit by bit basis.

The following figure shows the CCR bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



The following table describes the CCR bit assignments.

Table 4-22 CCR bit assignments without the Security Extension

Bits	Name	Function
[31:21]	-	Reserved, RES0
[20]	TRD	Thread reentrancy disabled. Enables checking for exception stack frame integrity signatures on SG instructions. If enabled this check causes a fault to be raised if an attempt is made to re-enter Secure Thread mode while a call to Secure Thread mode is already in progress. **0** Integrity signature checking not performed. **1** Integrity signature checking performed.
[19]	LOB	Loop and branch info cache enable. Enables the branch cache used by loop and branch future instructions for the selected Security state. **0** Branch cache disabled for the selected Security state. **1** Branch cache enabled for the selected Security state.
[18]	BP	RAZ/WI.
[17]	IC	Instruction cache enable. **0** Instruction caches disabled for the selected Security state. **1** Instruction caches enabled for the selected Security state.
[16]	DC	Data cache enable. **0** Data caching disabled. **1** Data caching enabled.
[15:11]	-	Reserved, RES0
[10]	STKOFHFNMIGN	Controls the effect of a stack limit violation while executing at a requested priority less than 0. **0** Stack limit faults not ignored. **1** Stack limit faults at requested priorities of less than 0 ignored.
[9]	-	Reserved, RES1.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-22 CCR bit assignments without the Security Extension (continued)

Bits	Name	Function
[8]	BFHFNMIGN	Determines the effect of precise bus faults on handlers running at a requested priority less than 0. **0** Precise bus faults are not ignored. **1** Precise bus faults at requested priorities of less than 0 are ignored.
[7:5]	-	Reserved, RES0.
[4]	DIV_0_TRP	Divide by zero trap. Controls the generation of a DIVBYZERO UsageFault when attempting to perform integer division by zero. **0** DIVBYZERO UsageFault generation disabled. **1** DIVBYZERO UsageFault generation enabled.
[3]	UNALIGN_TRP	Controls the trapping of unaligned word or halfword accesses. **0** Unaligned trapping disabled. **1** Unaligned trapping enabled.
[2]	-	Reserved, RES0.
[1]	USERSETMPEND	User set main pending. Determines whether unprivileged accesses are permitted to pend interrupts from the STIR. **0** Unprivileged accesses to the STIR generate a fault. **1** Unprivileged accesses to the STIR are permitted.
[0]	-	Reserved, RES1.

Table 4-23 CCR bit assignments with the Security Extension

Bits	Name	Function
[31:21]	-	Reserved, RES0
[20]	TRD	Thread reentrancy disabled. Enables checking for exception stack frame integrity signatures on SG instructions. If enabled this check causes a fault to be raised if an attempt is made to re-enter Secure Thread mode while a call to Secure Thread mode is already in progress. **0** Integrity signature checking not performed. **1** Integrity signature checking performed.
[19]	LOB	Loop and branch info cache enable. Enables the branch cache used by loop and branch future instructions for the selected Security state. **0** Branch cache disabled for the selected Security state. **1** Branch cache enabled for the selected Security state.
[18]	BP	RAZ/WI.
[17]	IC	Instruction cache enable. **0** Instruction caches disabled for the selected Security state. **1** Instruction caches enabled for the selected Security state.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-23 CCR bit assignments with the Security Extension (continued)

Bits	Name	Function
[16]	DC	Data cache enable. **0** Data caching disabled. **1** Data caching enabled.
[15:11]	-	Reserved, RES0
[10]	STKOFHFNMIGN	Controls the effect of a stack limit violation while executing at a requested priority less than 0. **0** Stack limit faults not ignored. **1** Stack limit faults at requested priorities of less than 0 ignored. This bit is banked between Security states.
[9]	-	Reserved, RES1.
[8]	BFHFNMIGN	Determines the effect of precise bus faults on handlers running at a requested priority less than 0. **0** Precise bus faults are not ignored. **1** Precise bus faults at requested priorities of less than 0 are ignored. This bit is not banked between Security states.
[7:5]	-	Reserved, RES0.
[4]	DIV_0_TRP	Divide by zero trap. Controls the generation of a DIVBYZERO UsageFault when attempting to perform integer division by zero. **0** DIVBYZERO UsageFault generation disabled. **1** DIVBYZERO UsageFault generation enabled. This bit is banked between Security states.
[3]	UNALIGN_TRP	Controls the trapping of unaligned word or halfword accesses. **0** Unaligned trapping disabled. **1** Unaligned trapping enabled. This bit is banked between Security states.
[2]	-	Reserved, RES0.
[1]	USERSETMPEND	User set main pending. Determines whether unprivileged accesses are permitted to pend interrupts from the STIR. **0** Unprivileged accesses to the STIR generate a fault. **1** Unprivileged accesses to the STIR are permitted. This bit is banked between Security states.
[0]	-	Reserved, RES1.

4.3.12 Configurable Fault Status Register

The CFSR indicates the cause of a MemManage fault, BusFault, or UsageFault.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CFSR attributes.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

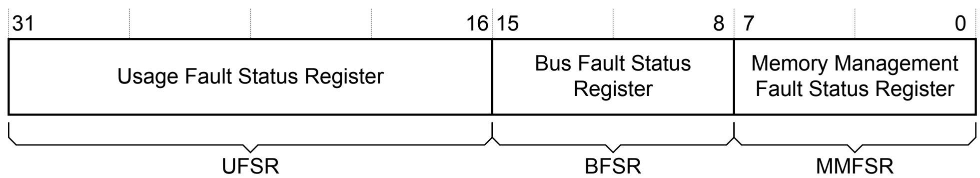
Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED28`.
Non-secure alias is provided using CFSR_NS, located at `0xE002ED28`.
If the Security Extension is implemented, Non-secure alias is provided using CFSR_NS, located at `0xE002ED28`.
This register is banked between Security states on a bit by bit basis.
If the Security Extension is implemented, this register is between Security states on a bit by bit basis.

The following figure shows the CFSR bit assignments.



The following table describes the CFSR bit assignments.

Table 4-24 CFSR register bit assignments

Bits	Name	Function	Security state
[31:16]	UFSR	*UsageFault Status Register* on page 4-492	Banked between Security states.
[15:8]	BFSR	*BusFault Status Register* on page 4-494	Not banked between Security states.
[7:0]	MMFSR	*MemManage Fault Status Register* on page 4-496	Banked between Security states.

The CFSR is byte accessible. You can access the CFSR or its subregisters as follows:

- Access the complete CFSR with a word access to `0xE000ED28`.
- Access the MMFSR with a byte access to `0xE000ED28`.
- Access the MMFSR and BFSR with a halfword access to `0xE000ED28`.
- Access the BFSR with a byte access to `0xE000ED29`.
- Access the UFSR with a halfword access to `0xE000ED2A`.

UsageFault Status Register

The UFSR is a subregister of the CFSR. The UFSR indicates the cause of a UsageFault.

Usage constraints

See *UsageFault Status Register* on page 4-492 for the CFSR attributes.

Configuration

This register is always implemented.

Attributes

A 15-bit RW register located at `0xE000ED2A`.
Non-secure alias is provided using UFSR_NS, located at `0xE002ED2A`.
If the Security Extension is implemented, the Non-secure alias is provided using UFSR_NS, located at `0xE002ED2A`.
This register is banked between Security states.
If the Security Extension is implemented, this register is banked between Security states on a bit by bit basis.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following figure shows the UFSR bit assignments.

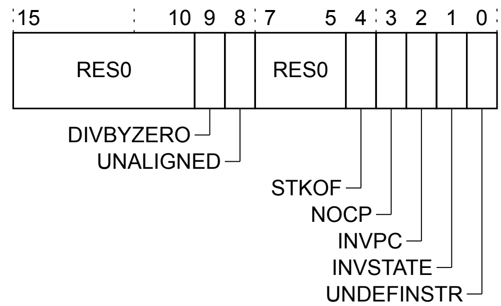


Figure 4-9 UFSR bit assignments

The following table describes the UFSR bit assignments.

Table 4-25 UFSR bit assignments

Bits	Name	Function
[15:10]	-	Reserved, RES0.
[9]	DIVBYZERO	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.
[8]	UNALIGNED	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.
[7:5]	-	Reserved, RES0.
[4]	STKOF	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.
[3]	NOCP	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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-25 UFSR bit assignments (continued)

Bits	Name	Function
[2]	INVPC	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.
[1]	INVSTATE	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.
[0]	UNDEFINSTR	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 bit resets to zero.

Note

All the bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

BusFault Status Register

The BFSR is a subregister of the CFSR. The flags in the BFSR indicate the cause of a bus access fault.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CFSR attributes.

Configuration

This register is always implemented.

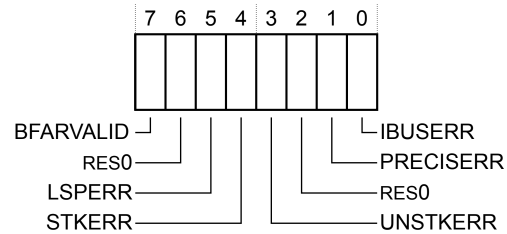
Attributes

A 8-bit RW register located at `0xE000ED29`.
Non-secure alias is provided using BFSR_NS, located at `0xE002ED29`.
If the Security Extension is implemented, the Non-secure alias is provided using BFSR_NS, located at `0xE002ED29`.
This register is not banked between Security states.
For an implementation with Security Extension:

- This field is not banked between Security states.
- If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

The following figure shows the BFSR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



The following table describes the BFSR bit assignments.

Table 4-26 BFSR bit assignments

Bits	Name	Function
[7]	BFARVALID	*BusFault Address Register* (BFAR) valid flag: **0** Value in BFAR is not a valid fault address. **1** BFAR holds a valid fault address. The processor sets this bit to 1 after a BusFault where the address is known. Other faults can set this bit to 0, such as a MemManage fault occurring later. If a BusFault occurs and is escalated to a hard fault because of priority, the hard fault handler must set this bit to 0. This prevents problems if returning to a stacked active BusFault handler whose BFAR value has been overwritten.
[6]	-	Reserved, RES0.
[5]	LSPERR	**0** No bus fault occurred during floating-point lazy state preservation. **1** A bus fault occurred during floating-point lazy state preservation.
[4]	STKERR	BusFault on stacking for exception entry: **0** No stacking fault. **1** Stacking for an exception entry has caused one or more BusFaults. When the processor sets this bit to 1, the SP is still adjusted but the values in the context area on the stack might be incorrect. The processor does not write a fault address to the BFAR.
[3]	UNSTKERR	BusFault on unstacking for a return from exception: **0** No unstacking fault. **1** Unstack for an exception return has caused one or more BusFaults. This fault is chained to the handler. This means that when the processor sets this bit to 1, the original return stack is still present. The processor does not adjust the SP from the failing return, does not performed a new save, and does not write a fault address to the BFAR.
[2]	-	Reserved, RES0

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-26 BFSR bit assignments (continued)

Bits	Name	Function
[1]	PRECISERR	Precise data bus error: **0** No precise data bus error. **1** A data bus error has occurred, and the PC value stacked for the exception return points to the instruction that caused the fault. When the processor sets this bit to 1, it writes the faulting address to the BFAR.
[0]	IBUSERR	Instruction bus error: **0** No instruction bus error. **1** Instruction bus error. The processor detects the instruction bus error on prefetching an instruction, but it sets the IBUSERR flag to 1 only if it attempts to issue the faulting instruction. When the processor sets this bit to 1, it does not write a fault address to the BFAR.

Note

The BFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

MemManage Fault Status Register

The MMFSR is a subregister of the CFSR. The flags in the MMFSR indicate the cause of memory access faults.

Usage constraints
See *4.3.1 System control block registers summary* on page 4-474 for the CFSR attributes.

Configuration
This register is always implemented.

Attributes
A 8-bit RW register located at `0xE000ED28`.
Non-secure alias is provided using MMFSR_NS, located at `0xE002ED28`.
If the Security Extension is implemented, the Non-secure alias is provided using MMFSR_NS, located at `0xE002ED28`.
This register is banked between Security states.
If the Security Extension is implemented, this register is between Security states on a bit by bit basis.

The following figure shows the MMFSR bit assignments.

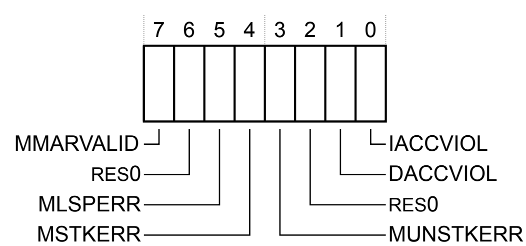


Figure 4-10 MMFSR bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table describes the MMFSR bit assignments.

Table 4-27 MMFSR bit assignments

Bits	Name	Function
[7]	MMARVALID	*MemManage Fault Address Register* (MMFAR) valid flag: **0** Value in MMFAR is not a valid fault address. **1** MMFAR holds a valid fault address. If a MemManage fault occurs and is escalated to a HardFault because of priority, the HardFault handler must set this bit to 0. This prevents problems on return to a stacked active MemManage fault handler whose MMFAR value has been overwritten.
[6]	-	Reserved, RES0.
[5]	MLSPERR	**0** No MemManage fault occurred during floating-point lazy state preservation. **1** A MemManage fault occurred during floating-point lazy state preservation.
[4]	MSTKERR	MemManage fault on stacking for exception entry: **0** No stacking fault. **1** Stacking for an exception entry has caused one or more access violations. When this bit is 1, the SP is still adjusted but the values in the context area on the stack might be incorrect. The processor has not written a fault address to the MMFAR.
[3]	MUNSTKERR	MemManage fault on unstacking for a return from exception: **0** No unstacking fault. **1** Unstack for an exception return has caused one or more access violations. This fault is chained to the handler. This means that when this bit is 1, the original return stack is still present. The processor has not adjusted the SP from the failing return, and has not performed a new save. The processor has not written a fault address to the MMFAR.
[2]	-	Reserved, RES0.
[1]	DACCVIOL	Data access violation flag: **0** No data access violation fault. **1** The processor attempted a load or store at a location that does not permit the operation. When this bit is 1, the PC value stacked for the exception return points to the faulting instruction. The processor has loaded the MMFAR with the address of the attempted access.
[0]	IACCVIOL	Instruction access violation flag: **0** No instruction access violation fault. **1** The processor attempted an instruction fetch from a location that does not permit execution. This fault occurs on any access to an XN region, even when the MPU is disabled or not present. When this bit is 1, the PC value stacked for the exception return points to the faulting instruction. The processor has not written a fault address to the MMFAR.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The MMFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

4.3.13 Coprocessor Access Control Register

The CPACR register specifies the access privileges for coprocessors.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CPACR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED88`.
Non-secure alias is provided using CPACR_NS, located at `0xE002ED88`.
If the Security Extension is implemented, Non-secure alias is provided using CFSR_NS, located at `0xE002ED28`.
This register is banked between Security states on a bit by bit basis.
If the Security Extension is implemented, this register is banked between Security states on a bit by bit basis.

The following figure shows the CPACR bit assignments.

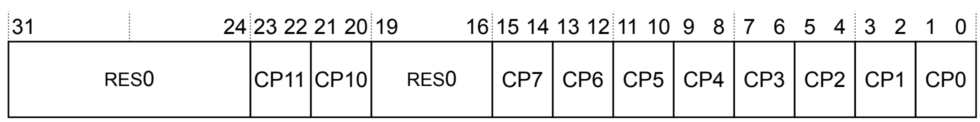


Figure 4-11 CPACR bit assignments

The following table describes the CPACR bit assignments.

Table 4-28 CPACR bit assignments

Bits	Name	Function
[31:24]	-	Reserved, RES0
[23:22]	CP11	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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-28 CPACR bit assignments (continued)

Bits	Name	Function
[21:20]	CP10	CP10 Privilege. Defines the access rights for the floating-point functionality. The possible values of this bit 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 floating-point instruction and access to any floating-point registers D0-D16. If the implementation does not include the FP Extension, this field is RAZ/WI.
[19:16]	-	Reserved, RES0
CP*m*, bits[2*m* +1:2*m*], for *m* = 0-7	CP*m*	Coprocessor *m* privilege. Controls access privileges for coprocessor *m*. The possible values of this bit 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.

4.3.14 Processor Feature Register 0

The ID_PFR0 register contains a field that indicates the version of the *Reliability, Availability, and Serviceability* (RAS) extension supported.

Usage constraints

Unprivileged access results in a BusFault exception. For more information on ID_PFR0 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED40`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_PFR0_NS is located at `0xE002ED40`.
Non-secure alias is provided using ID_PFR0_NS is located at `0xE002ED40`.

The following figure shows the ID_PFR0 bit assignments.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

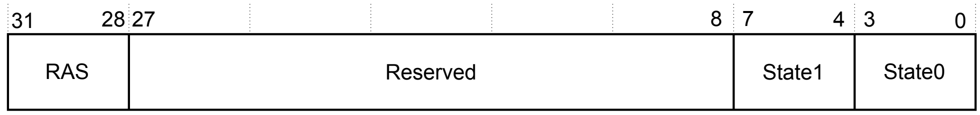


Figure 4-12 ID_PFR0 bit assignments

The following table describes the ID_PFR0 bit assignments.

Table 4-29 ID_PFR0 bit assignments

Field	Name	Type	Description
[31:28]	RAS	RO	Identifies which version of the RAS architecture is implemented. `0b0010` Version 1.
[27:8]	Reserved	-	RES0
[7:4]	State1	RO	T32 instruction set support. `0b0011` T32 instruction set including Thumb-2 technology is implemented.
[3:0]	State0	RO	A32 instruction set support. `0b0000` A32 instruction set is not implemented.

4.3.15 Processor Feature Register 1

The ID_PFR1 register gives information about the programmers' model and Extensions support.

Usage constraints

Privileged access permitted only. For more information on ID_PFR1 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED44`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_PFR1_NS is located at `0xE002ED44`.
Non-secure alias is provided using ID_PFR0_NS is located at `0xE002ED44`.

The following figure shows the ID_PFR1 bit assignments.

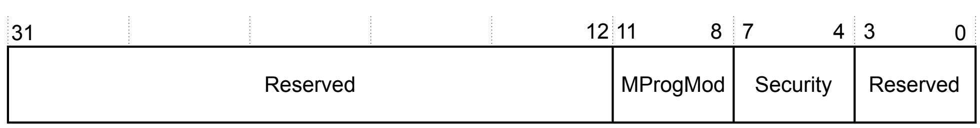


Figure 4-13 ID_PFR1 bit assignments

The following table describes the ID_PFR1 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-30 ID_PFR1 bit assignments

Field	Name	Type	Description
[31:12]	Reserved	-	RES0
[11:8]	MProgMod	RO	M-profile programmers' model. Identifies support for the M-profile programmers' model. `0b0010` Two-stack programmers' model
[7:4]	Security	RO	Security. Identifies whether the Security Extension in implemented. `0b0000` Security Extension is not implemented. `0b0011` Security Extension implemented with state handling instructions (VSCCLRM, CLRM, FPCXT access instructions and disabling SG Thread mode re-entrancy).
[3:0]	Reserved	-	RES0

4.3.16 Debug Fault Status Register

The DFSR shows which debug event has occurred.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.
The register is accessible to accessed through unprivileged *Debug Access Port* (DAP) requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED30`.
Non-secure alias is provided using DFSR_NS, located at `0xE002ED30`.
A 32-bit RW register located at `0xE000ED30`. If the Security Extension is implemented, non-secure alias is provided using DFSR_NS, located at `0xE002ED30`.
This register is not banked between Security states.
If the Security Extension is implemented, this register is not banked between Security states.

The following figure shows the DFSR bit assignments.

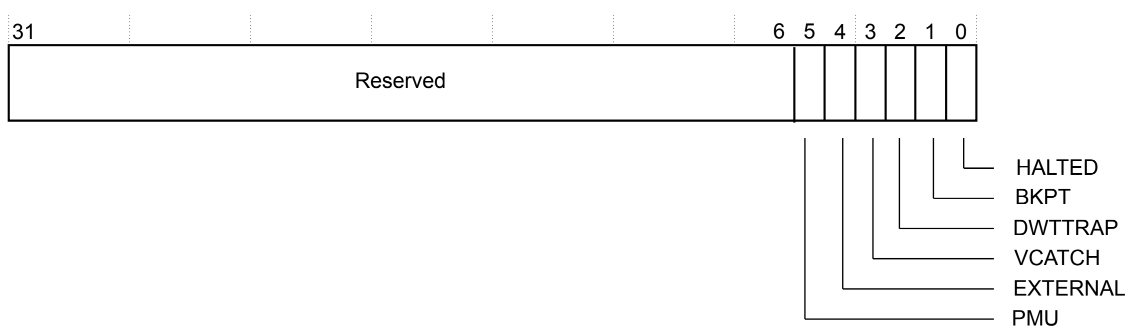


Figure 4-14 DFSR bit assignments

The following table describes the DFSR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-31 DFSR bit assignments

Bits	Name	Type	Description
[31:6]	-	-	Reserved, RES0
[5]	PMU	RW	PMU event. Sticky flag indicating whether a PMU counter overflow event has occurred. The possible values are: **0** PMU event has not occurred. **1** PMU event has occurred.
[4]	EXTERNAL	-	External event. Sticky flag indicating whether an External debug event has occurred. The possible values are: **0** Debug event has not occurred. **1** Debug event has occurred.
[3]	VCATCH	RW	Vector Catch event. Sticky flag indicating whether a Vector Catch debug event has occurred. The possible values are: **0** Debug event has not occurred. **1** Debug event has occurred.
[2]	DWTTRAP	RW	Watchpoint event. Sticky flag indicating whether a Watchpoint debug event has occurred. The possible values are: **0** Debug event has not occurred. **1** Debug event has occurred.
[1]	BKPT	RW	Breakpoint event. Sticky flag indicating whether a Breakpoint debug event has occurred. The possible values are: **0** Debug event has not occurred. **1** Debug event has occurred.
[0]	HALTED	RW	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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.17 HardFault Status Register

The HFSR gives information about events that activate the HardFault handler. The HFSR register is read, write to clear. This means that bits in the register read normally, but writing 1 to any bit clears that bit to 0.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the HFSR attributes.

Configuration

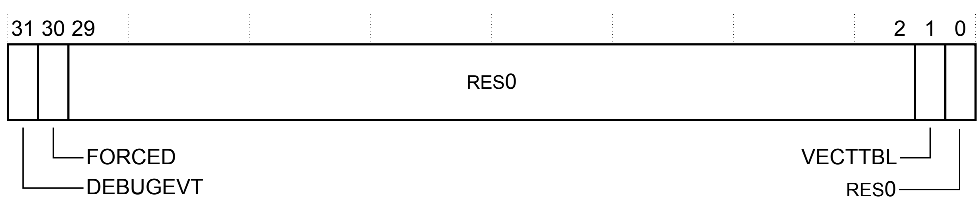
This register is always implemented.

Attributes

A 8-bit RW register located at `0xE000ED2C`.
This register is not banked between Security states.
If the Security Extension is implemented, the Non-secure alias is provided using HFSR_NS, located at `0xE002ED2C`.
The Non-secure alias is provided using HFSR_NS, located at `0xE002ED2C`.
In an implementation with the Security Extension:

- This field is not banked between Security states.
- If AIRCR.BFHFNMINS is zero this field is RAZ/WI from Non-secure state.

The following figure shows the HFSR bit assignments.



The following table describes the HFSR bit assignments.

Table 4-32 HFSR bit assignments

Bits	Name	Function
[31]	DEBUGEVT	Reserved for Debug use. When writing to the register you must write 1 to this bit, otherwise behavior is UNPREDICTABLE.
[30]	FORCED	Indicates a forced HardFault, generated by escalation of a fault with configurable priority that cannot be handled, either because of priority or because it is disabled: **0** No forced HardFault. **1** Forced HardFault. When this bit is set to 1, the HardFault handler must read the other fault status registers to find the cause of the fault.
[29:2]	-	Reserved, RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-32 HFSR bit assignments (continued)

Bits	Name	Function
[1]	VECTTBL	Indicates a HardFault on a vector table read during exception processing: **0** No HardFault on vector table read. **1** HardFault on vector table read. This error is always handled by the HardFault handler. When this bit is set to 1, the PC value stacked for the exception return points to the instruction that was pre-empted by the exception.
[0]	-	Reserved, RES0.

Note

The HFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

4.3.18 Interrupt Control and State Register

The ICSR provides a set-pending bit for the non-maskable interrupt exception, and set-pending and clear-pending bits for the PendSV and SysTick exceptions.

The ICSR indicates:

- The exception number of the exception being processed.
- Whether there are pre-empted active exceptions.
- The exception number of the highest priority pending exception
- Whether any interrupts are pending.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the ICSR attributes.

Configuration

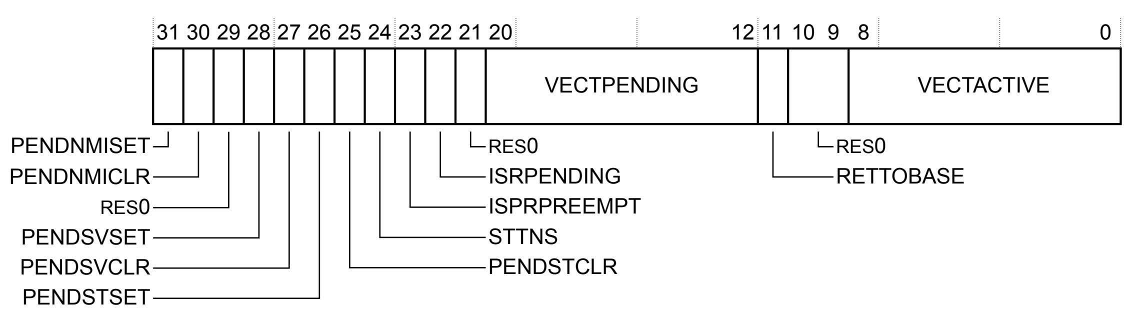
This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED00`. Non-secure alias is provided using CPUID_NS, located at `0xE002ED00`.
This register is banked between Security states on a bit by bit basis.

In an implementation with the Security Extension, this register is banked between Security states on a bit by bit basis.

The following figure shows the ICSR bit assignments.



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table shows the ICSR bit assignments.

Table 4-33 ICSR bit assignments with the Security Extension

Bits	Name	Type	Function
[31]	PENDNMISET	RW	NMI set-pending bit. Write: **0** No effect. **1** Changes NMI exception state to pending. Read: **0** NMI exception is not pending. **1** NMI exception is pending. A read of this bit by the NMI exception handler returns 1 only if the NMI signal is reasserted while the processor is executing that handler.
[30]	PENDNMICLR	WO	Pend NMI clear bit. Write: **0** No effect. **1** Clear pending status. 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.
[29]	-	-	Reserved, RES0.
[28]	PENDSVSET	RW	PendSV set-pending bit. Write: **0** No effect. **1** Changes PendSV exception state to pending. Read: **0** PendSV exception is not pending. **1** PendSV exception is pending. Writing 1 to this bit is the only way to set the PendSV exception state to pending. This bit is banked between Security states.
[27]	PENDSVCLR	WO	PendSV clear-pending bit. Write: **0** No effect. **1** Removes the pending state from the PendSV exception. This bit is banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-33 ICSR bit assignments with the Security Extension (continued)

Bits	Name	Type	Function
[26]	PENDSTSET	RW	SysTick exception set-pending bit. Write: **0** No effect. **1** Changes SysTick exception state to pending. Read: **0** SysTick exception is not pending. **1** SysTick exception is pending. This bit is banked between Security states.
[25]	PENDSTCLR	WO	SysTick exception clear-pending bit. Write: **0** No effect. **1** Removes the pending state from the SysTick exception. This bit is WO. On a register read, its value is UNKNOWN. This bit is not banked between Security states.
[24]	STTNS	RO	Reserved, RES0.
[23]	ISRPREEMPT	RO	Interrupt preempt. Indicates whether a pending exception is handled on exit from debug state. The possible values of this bit are: **0** Pending exception is not handled on exit from debug state. **1** Pending exception is handled on exit from debug state. This field is not banked between Security states.
[22]	ISRPENDING	RO	Interrupt pending flag, excluding NMI and Faults: **0** Interrupt not pending. **1** Interrupt pending. This bit is not banked between Security states.
[21]	-	-	Reserved, RES0.
[20:12]	VECTPENDING	RO	Indicates the exception number of the highest priority pending enabled exception: **0** No pending exceptions. **Nonzero** The exception number of the highest priority pending enabled exception. The value that this field indicates includes the effect of the BASEPRI and FAULTMASK registers, but not any effect of the PRIMASK register. This field is not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-33 ICSR bit assignments with the Security Extension (continued)

Bits	Name	Type	Function
[11]	RETTOBASE	RO	Indicates whether there are pre-empted active exceptions: **0** There are pre-empted active exceptions to execute. **1** There are no active exceptions, or the currently executing exception is the only active exception. This bit is not banked between Security states.
[10:9]	-	-	Reserved, RES0.
[8:0]	VECTACTIVE **Note** This is the same value as IPSR bits[8:0]. For more information, see *Interrupt Program Status Register* on page 2-40	RO	Contains the active exception number: **0** Thread mode. **1** The exception number of the currently active exception. **Note** Subtract 16 from this value to obtain the CMSIS IRQ number required to index into the Interrupt Clear-Enable, Set-Enable, Clear-Pending, Set-Pending, or Priority Registers, see *Interrupt Program Status Register* on page 2-40. This field is not banked between Security states.

When you write to the ICSR, the effect is UNPREDICTABLE if you:
- Write 1 to the PENDSVSET bit and write 1 to the PENDSVCLR bit.
- Write 1 to the PENDSTSET bit and write 1 to the PENDSTCLR bit.

4.3.19 Instruction Set Attribute Register 0

The ID_ISAR0 register gives information about the implemented instruction set.

Usage constraints

Privileged access permitted only. For more information on ID_ISAR0 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED60`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_ISAR0_NS is located at `0xE002ED60`.
Non-secure alias is provided using ID_ISAR0_NS is located at `0xE002ED60`.

If coprocessors excluding the floating-point and MVE functionality are not supported, this register reads as `0x01101110`

If coprocessors excluding the floating-point and MVE functionality are supported, this register reads as `0x01141110`

The following figure shows the ID_ISAR0 bit assignments.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

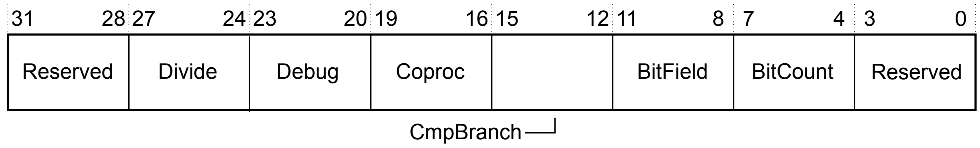


Figure 4-15 ID_ISAR0 bit assignments

The following table describes the ID_ISAR0 bit assignments.

Table 4-34 ID_ISAR0 bit assignments

Field	Name	Type	Description
[31:28]	Reserved	RO	RES0
[27:24]	Divide	RO	Divide. Indicates the supported divide instructions. `0b0001` Supports `SDIV` and `UDIV` instructions.
[23:20]	Debug	RO	Debug. Indicates the implemented debug instructions. `0b0001` Supports `BKPT` instruction.
[19:16]	Coproc	RO	Coprocessor. Indicates the supported coprocessor instructions. The possible values are: `0b0000` Coprocessor instructions are not supported, except for, floating-point or MVE. `0b0100` Coprocessor instructions are supported.
[15:12]	CmpBranch	RO	Compare and branch. Indicates the supported combined Compare and Branch instructions. `0b0011` Supports `CBNZ` and `CBZ` instructions along with non-predicated low-overhead looping (`WLS`, `DLS`, `LE`, and `LCTP`) and branch future (`BF`, `BFX`, `BFL`, `BFLX`, and `BFCSEL`) instructions. Branch future instructions behave as NOPs.
[11:8]	BitField	RO	Bit field. Indicates the supported bit field instructions. `0b0001` `BFC`, `BFI`, `SBFX`, and `UBFX` are supported.
[7:4]	BitCount	RO	Bit count. Indicates the supported bit count instructions. `0b0001` `CLZ` supported.
[3:0]	Reserved	-	RES0

4.3.20 Instruction Set Attribute Register 1

The ID_ISAR1 register gives information about the implemented instruction set.

Usage constraints

Privileged access permitted only. For more information on ID_ISAR1 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

32-bit RO register located at `0xE000ED64`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_ISAR1_NS is located at `0xE002ED64`.
Non-secure alias is provided using ID_ISAR1_NS is located at `0xE002ED64`.

The following figure shows the ID_ISAR1 bit assignments.

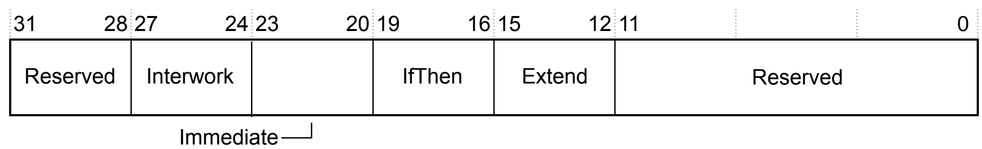


Figure 4-16 ID_ISAR1 bit assignments

The following table describes the ID_ISAR1 bit assignments.

Table 4-35 ID_ISAR1 bit assignments

Field	Name	Type	Description
[31:28]	Reserved	-	RES0
[27:24]	Interwork	RO	Interworking. Indicates the implemented interworking instructions. `0b0010` `BLX`, `BX`, and loads to PC network.
[23:20]	Immediate	RO	Immediate. Indicates the implemented data-processing instructions with long immediates. `0b0010` `ADDW`, `MOVW`, `MOVT`, and `SUBW` are supported.
[19:16]	IfThen	RO	If-Then. Indicates the implemented If-Then instructions. `0b0001` `IT` instruction supported.
[15:12]	Extend	RO	Extend. Indicates the implemented Extend instructions. `0b0010` Adds `SXTB16`, `SXTAB`, `SXTAB16`, `SXTAH`, `UXTB16`, `UXTAB`, `UXTAB16`, and `UXTAH` DSP Extension only.
[11:0]	Reserved	-	RES0

4.3.21 Instruction Set Attribute Register 2

The ID_ISAR2 register gives information about the implemented instruction set.

Usage constraints

Privileged access permitted only. For more information on ID_ISAR2 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

32-bit RO register located at `0xE000ED68`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_ISAR2_NS is located at `0xE002ED68`.
Non-secure alias is provided using ID_ISAR2_NS is located at `0xE002ED68`.

The following figure shows the ID_ISAR2 bit assignments.

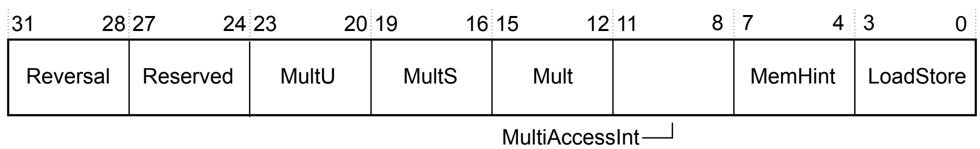


Figure 4-17 ID_ISAR2 bit assignments

The following table describes the ID_ISAR2 bit assignments.

Table 4-36 ID_ISAR2 bit assignments

Field	Name	Type	Description
[31:28]	Reversal	RO	Reversal. Indicates the implemented reversal instructions. `0b0010` `REV`, `REV16`, `REVSH`, and `RBIT` instructions are supported.
[27:24]	Reserved	-	RES0
[23:20]	MultU	RO	Multiply unsigned. Indicates the implemented Advanced Unsigned Multiple instructions. `0b0010` Adds `UMAAL`, DSP Extension only.
[19:16]	MultS	RO	Multiply signed. Indicates the implemented Advanced Signed Multiple instructions. **`0b0011`** Adds all saturating and DSP signed multiplies, DSP Extension only.
[15:12]	Mult	RO	Multiplies.Indicates the implemented additional Multiply instructions. `0b0010` `MUL`, `MLA`, and `MLS` instructions.
[11:8]	MultiAccessInt	RO	Multi-access instructions. Indicates the support for interruptible multi-access instructions. **`0b0010`** `LDM`, `STM`, and `CLRM` instructions are continuable.
[7:4]	MemHint	RO	Memory hint. Indicates the implemented memory hint instructions. **`0b0011`** `PLI` and `PLD` instructions implemented.
[3:0]	LoadStore	RO	Load/store. Indicates the implemented additional load/store instructions. **`0b0010`** Supports load-acquire, store-release, and exclusive load and store instructions.

4.3.22 Instruction Set Attribute Register 3

The ID_ISAR3 register gives information about the implemented instruction set.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints

Privileged access permitted only. For more information on ID_ISAR3 attributes, see *4.3.1 System control block registers summary* on page 4-474.

This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED68`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_ISAR3_NS is located at `0xE002ED68`.
Non-secure alias is provided using ID_ISAR3_NS is located at `0xE002ED68`.

The following figure shows the ID_ISAR3 bit assignments.

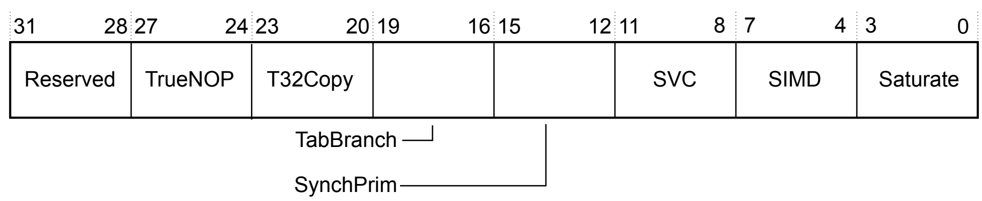


Figure 4-18 ID_ISAR3 bit assignments

The following table describes the ID_ISAR3 bit assignments.

Table 4-37 ID_ISAR3 bit assignments

Field	Name	Type	Description
[31:28]	Reserved	-	RES0
[27:24]	TrueNOP	RO	True no-operation. Indicates the implemented true `NOP` instructions. `0b0001` `NOP` instruction and compatible hints are implemented.
[23:20]	T32Copy	RO	T32 copy. Indicates the support for T32 non-flag setting `MOV` instructions `0b0001` Encoding T1 of MOV (register) supports copying a low register to a low register. Low registers are general purpose registers in the range R0-R7.
[19:16]	TabBranch	RO	Table branch. Indicates the implemented table branch instructions. **`0b0001`** `TBB` and `TBH` are implemented.
[15:12]	SynchPrim	RO	Synchronization primitives. Used with ID_ISAR4.SynchPrim_frac to indicate the implemented synchronization primitive instructions. `0b0001` `LDREX`, `STREX`, `LDREXB`, `STREXB`, `LDREXH`, `STREXH`, and `CLREX` are implemented
[11:8]	SVC	RO	Supervisor Call. Indicates the implemented SVC instructions. **`0b0001`** `SVC` instruction implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-37 ID_ISAR3 bit assignments (continued)

Field	Name	Type	Description
[7:4]	SIMD	RO	Single-instruction, multiple-data. Indicates the implemented SIMD instructions. **`0b0011`** `SSAT`, `USAT`, Q-bit, packed arithmetic, and GE-bits are implemented.
[3:0]	Saturate	RO	Saturate. Indicates the implemented saturating instructions. **`0b0001`** `QADD`, `QDADD`, `QDSUB`, `QSUB`, and Q-bit implemented, DSP Extension only.

4.3.23 Instruction Set Attribute Register 4

The ID_ISAR4 register gives information about the implemented instruction set.

Usage constraints

Privileged access permitted only. For more information on ID_ISAR4 attributes, see *4.3.1 System control block registers summary* on page 4-474.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

32-bit RO register located at `0xE000ED68`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_ISAR4_NS is located at `0xE002ED68`.
Non-secure alias is provided using ID_ISAR4_NS is located at `0xE002ED68`.

The following figure shows the ID_ISAR4 bit assignments.

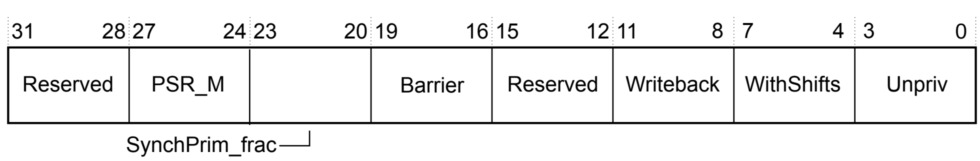


Figure 4-19 ID_ISAR4 bit assignments

The following table describes the ID_ISAR4 bit assignments.

Table 4-38 ID_ISAR4 bit assignments

Field	Name	Type	Description
[31:28]	Reserved	-	RES0
[27:24]	PSR_M	RO	Program Status Registers M. Indicates the implemented M-profile instructions to modify the PSRs. `0b0001` M-profile forms of `CPS`, `MRS`, and `MSR` instructions are implemented.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-38 ID_ISAR4 bit assignments (continued)

Field	Name	Type	Description
[23:20]	SynchPrim_frac	RO	Synchronization primitives fractional. Used in conjunction with ID_ISAR3.SynchPrim to indicate the implemented synchronization primitive instructions. `0b0011` `LDREX`, `STREX`, `CLREX`, `LDREXB`, `LDREXH`, `STREXB`, and `STREXH` instructions are implemented.
[19:16]	Barrier	RO	Barrier. Indicates the implemented Barrier instructions. **`0b0001`** `CSDB`, `DMB`, `DSB`, `ISB`, `PSSBB`, and `SSBB` barrier instructions implemented.
[15:12]	Reserved	-	RES0
[11:8]	Writeback	RO	Writeback. Indicates the support for writeback addressing modes. **`0b0001`** All writeback addressing modes supported.
[7:4]	WithShifts	RO	With shifts. Indicates the support for write-back addressing modes. **`0b0011`** Support for constant shifts on load/store and other instructions.
[3:0]	Unpriv	RO	Unprivileged. Indicates the implemented unprivileged instructions. **`0b0010`** `LDRBT`, `LDRHT`, `LDRSBT`, `LDRSHT`, `LDRT`, `STRBT`, `STRHT`, and `STRT` implemented.

4.3.24 Instruction Set Attribute Register 5

The ID_ISAR5 gives information about the implemented instruction set. This register is Reserved, RES0.

4.3.25 MemManage Fault Address Register

The MMFAR shows the address of the memory location that caused a *Memory Protection Unit* (MPU) fault.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
A 32-bit RW register located at `0xE000ED34`.
Non-secure alias is provided using MMFAR_NS, located at `0xE002ED34`.
A 32-bit RW register located at `0xE000ED34`. If the Security Extension is implemented, non-secure alias is provided using MMFAR_NS, located at `0xE002ED34`.
This register is not banked between Security states.
If the Security Extension is implemented, this register is not banked between Security states.

The following figure shows the MMFAR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 4-20 MMFAR bit assignments

The following table describes the MMFAR bit assignments.

Table 4-39 MMFAR bit assignments

Bits	Name	Type	Description
[31:0]	ADDRESS	RW	Data address for a MemManage fault. This field is valid only when MMFSR.MMARVALID is set, otherwise it is UNKNOWN.

4.3.26 Memory Model Feature Register 0

The ID_MMFR0 provides information about the implemented memory model and memory management support.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
32-bit RO register located at `0xE000ED50`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_MMFR0_NS is located at `0xE002ED50`.
Non-secure alias is provided using ID_MMFR0_NS is located at `0xE002ED50`.

The following figure shows the ID_MMFR0 bit assignments.

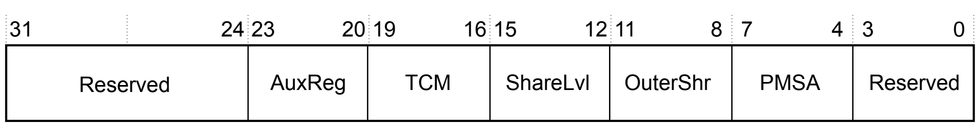


Figure 4-21 ID_MMFR0 bit assignments

The following table describes the ID_MMFR0 bit assignments.

Table 4-40 ID_MMFR0 bit assignments

Bits	Name	Type	Description
[31:24]	Reserved	-	RES0
[23:20]	AuxReg	RO	Auxiliary Registers. Indicates support for Auxiliary Control registers. `0b0001` Auxiliary Control registers are supported.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-40 ID_MMFR0 bit assignments (continued)

Bits	Name	Type	Description
[19:16]	TCM	RO	Tightly Coupled Memories. Indicates support for TCMs. `0b0001` TCMs are supported.
[15:12]	ShareLvl	RO	Shareability levels. Indicates the number of Shareability levels implemented. `0b0001` Two levels of Shareability implemented.
[11:8]	OuterShr	RO	Outermost Shareability. Indicates the outermost Shareability domain implemented. `0b0000` Implemented as Non-cacheable.
[7:4]	PMSA	RO	Protected memory system architecture. Indicates support for the protected memory system architecture (PMSA). `0b0100` Supports PMSAv8.
[3:0]	Reserved	-	RES0

4.3.27 Memory Model Feature Register 1

The ID_MMFR1 provides information about the IMPLEMENTATION DEFINED memory model and memory management support. This register is Reserved, RES0.

4.3.28 Memory Model Feature Register 2

The ID_MMFR2 provides information about the implemented memory model and memory management support.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
32-bit RO register located at `0xE000ED58`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_MMFR2_NS is located at `0xE002ED58`.
Non-secure alias is provided using ID_MMFR2_NS is located at `0xE002ED58`.

The following figure shows the ID_MMFR2 bit assignments.

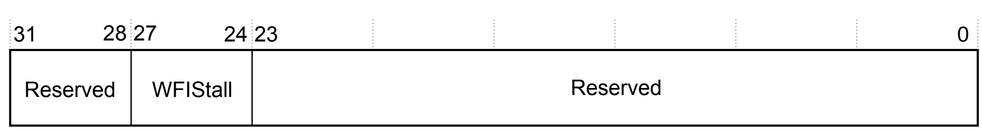


Figure 4-22 ID_MMFR2 bit assignments

The following table describes the ID_MMFR2 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-41 ID_MMFR2 bit assignments

Bits	Name	Type	Description
[31:28]	Reserved	-	RES0
[27:24]	WFIStall	RO	WFI stall. Indicates the support for *Wait For Interrupt* (WFI) stalling. The possible values are: `0b0001` WFI has the ability to stall.
[23:0]	Reserved	-	RES0

4.3.29 Memory Model Feature Register 3

The ID_MMFR3 provides information about the implemented memory model and memory management support.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
32-bit RO register located at `0xE000ED5C`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using ID_MMFR3_NS is located at `0xE002ED5C`.
Non-secure alias is provided using ID_MMFR3_NS is located at `0xE002ED5C`.

The following figure shows the ID_MMFR3 bit assignments.

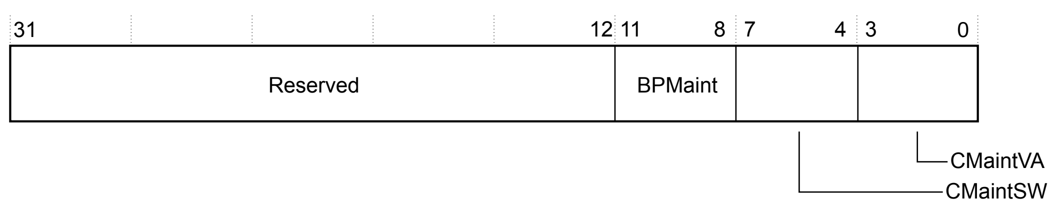


Figure 4-23 ID_MMFR3 bit assignments

The following table describes the ID_MMFR3 bit assignments.

Table 4-42 ID_MMFR3 bit assignments

Bits	Name	Type	Description
[31:12]	Reserved	-	RES0
[11:8]	BPMaint	RO	Branch predictor maintenance. Indicates the supported branch predictor maintenance. `0b0000` Branch predictor maintenance is not supported.
[7:4]	CMaintSW	RO	Cache maintenance set/way. Indicates the supported cache maintenance operations by set/way. `0b0001` Cache maintenance operations by set/way are supported.
[3:0]	CMaintVA	RO	Cache maintenance by address. Indicates the supported cache maintenance operations by address. `0b0001` Cache maintenance operations by address are supported.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.30 Non-secure Access Control Register

In an implementation with the Security Extension, the NSACR register defines the Non-secure access permissions for both the Floating-point and MVE and coprocessors CP *m*, bit[*m*], for *m* = 0-7. If MVE is implemented, this register specifies the Non-secure permissions for MVE.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the CPACR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED8C`.

If the Security Extension is implemented, this register is not banked between Security states and returns a value of `0x00000CFF`.

The following figure shows the NSACR bit assignments.

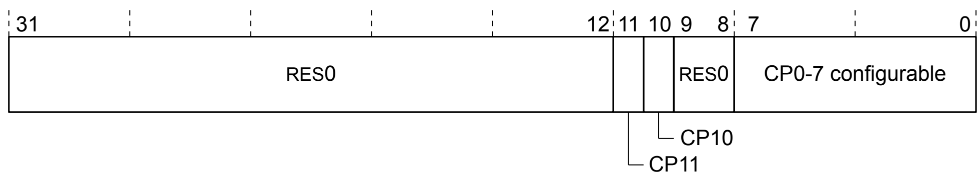


Figure 4-24 NSACR bit assignments

Table 4-43 NSACR bit assignments

Bits	Name	Function
[31:12]	-	Reserved, RES0.
[11]	CP11	CP11 access. Enables Non-secure access to the Floating-point and MVE. Programming with a different value other than that used for CP10 is UNPREDICTABLE. If the Floating-point and MVE are not implemented, this bit is RAZ/WI.
[10]	CP10	CP10 access. Enables Non-secure access to the Floating-point and MVE. **0** Non-secure accesses to the Floating-point Extension generate a NOCP UsageFault. **1** Non-secure access to the Floating-point and MVE permitted. If the Floating-point and MVE is not implemented, this bit is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-43 NSACR bit assignments (continued)

Bits	Name	Function
[9:8]	-	Reserved, RES0
CP*m*, bit[*m*], for *m* = 0-7 [7-0]	CP*m* for *m* = 0-7 CP	Access to CP*m*. Enables Non-secure access to coprocessor CP*m*: **0** Non-secure accesses to this coprocessor generate a NOCP UsageFault. **1** Non-secure access to this coprocessor permitted. If the CP*m* is not implemented, this bit is RAZ/WI.

4.3.31 System Control Register

The SCR controls features of entry to and exit from low-power state.

See *4.3.1 System control block registers summary* on page 4-474 for the SCR attributes.

32-bit RO register located at `0xE000ED10`.

This register is banked between Security states on a bit by bit basis.

In an implementation with the Security Extension, this register is banked between Security states on a bit by bit basis.

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.

If the Security Extension is implemented, 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.

The bit assignments are:

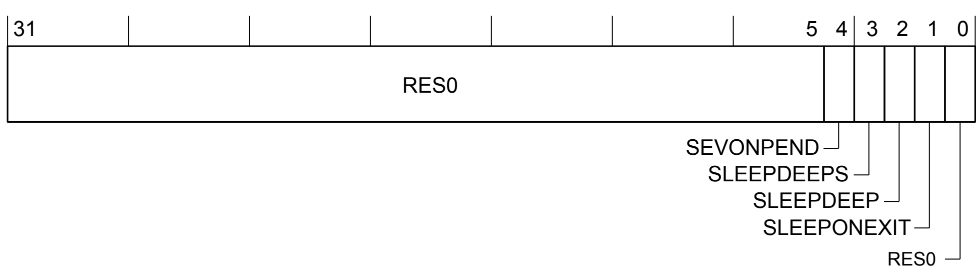


Figure 4-25 SCR bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-44 SCR bit assignments without the Security Extension

Bits	Name	Function
[31:5]	-	Reserved, RES0.
[4]	SEVONPEND	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. **0** Transitions from inactive to pending are not wakeup events. **1** Transitions from inactive to pending are wakeup events.
[3]	SLEEPDEEPS	RAZ/WI.
[2]	SLEEPDEEP	Indicates whether the processor uses sleep or deep sleep as its low-power mode: <Licensee to indicate their system-specific behavior> **0** Sleep. **1** Deep sleep.
[1]	SLEEPONEXIT	Indicates sleep-on-exit when returning from Handler mode to Thread mode: **0** Do not sleep when returning to Thread mode. **1** Enter sleep, or deep sleep, on return from an ISR. Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application.
[0]	-	Reserved, RES0.

Table 4-45 SCR bit assignments with the Security Extension

Bits	Name	Function
[31:5]	-	Reserved, RES0.
[4]	SEVONPEND	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. **0** Transitions from inactive to pending are not wakeup events. **1** Transitions from inactive to pending are wakeup events. This bit is banked between Security states.
[3]	SLEEPDEEPS	Controls whether the SLEEPDEEP bit is only accessible from the Secure state: **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 in only accessible from the Secure state, and behaves as RAZ/WI when accessed from the Non-secure state. This bit is not banked between Security states.
[2]	SLEEPDEEP	Controls whether the processor uses sleep or deep sleep as its low-power mode: <Licensee to indicate their system-specific behavior> **0** Sleep. **1** Deep sleep. This bit is not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-45 SCR bit assignments with the Security Extension (continued)

Bits	Name	Function
[1]	SLEEPONEXIT	Indicates sleep-on-exit when returning from Handler mode to Thread mode: <Licensee to indicate their system-specific behavior> **0** Do not sleep when returning to Thread mode. **1** Enter sleep, or deep sleep, on return from an ISR. Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application. This bit is banked between Security states.
[0]	-	Reserved, RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.32 System Handler Control and State Register

The SHCSR enables the system handlers. It indicates the pending status of the BusFault, MemManage fault, and SVC exceptions, and indicates the active status of the system handlers.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the SHCSR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED24`.
Non-secure alias is provided using SHCSR_NS, located at `0xE002ED24`.
A 32-bit RW register located at `0xE000ED20`. Non-secure alias is provided using SHCSR_NS, located at `0xE002ED20`.
This register is banked between Security states on a bit by bit basis.
If the Security Extension is implemented, this register is between Security states on a bit by bit basis.

The following figure shows the SHCSR bit assignments.

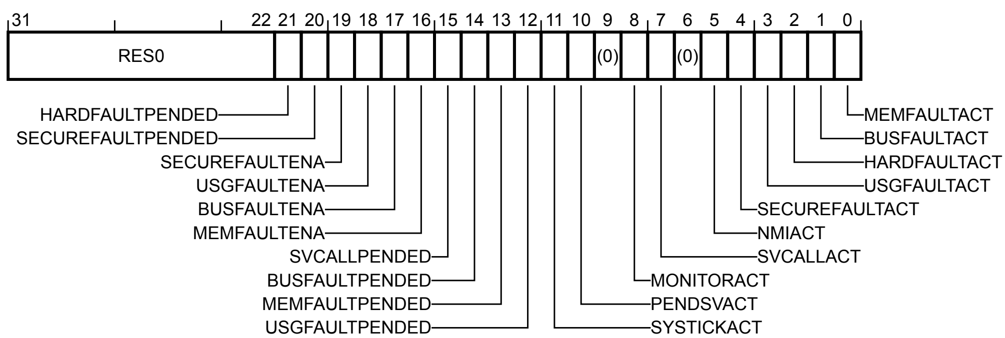


Figure 4-26 SHCSR bit assignments

The following table describes the SHCSR bit assignments.

Table 4-46 SHCSR bit assignments without the Security Extension

Bits	Name	Function
[31:22]	-	Reserved, RES0.
[21]	HARDFAULTPENDED	HardFault exception pended state bit, set to 1 to allow exception modification [n]
[20]	SECUREFAULTPENDED	RES0[n]
[19][31:22]	SECUREFAULTENA	RES0
[31:22[18]	USGFAULTENA	UsageFault enable bit, set to 1 to enable.[m]
[17]	BUSFAULTENA	BusFault enable bit, set to 1 to enable.[m]
[16]	MEMFAULTENA	MemManage enable bit, set to 1 to enable. [m]
[15]	SVCALLPENDED	SVCall pending bit, reads as 1 if exception is pending. [n]
[14]	BUSFAULTPENDED	BusFault exception pending bit, reads as 1 if exception is pending. [n]
[13]	MEMFAULTPENDED	MemManage exception pending bit, reads as 1 if exception is pending.[n]
[12]	USGFAULTPENDED	UsageFault exception pending bit, reads as 1 if exception is pending.[n]

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-46 SHCSR bit assignments without the Security Extension (continued)

Bits	Name	Function
[11]	SYSTICKACT	SysTick exception active bit, reads as 1 if exception is active.[o]
[10]	PENDSVACT	PendSV exception active bit, reads as 1 if exception is active
[9]	-	Reserved, RES0.
[8]	MONITORACT	Debug monitor active bit, reads as 1 if Debug monitor is active
[7]	SVCALLACT	SVCall active bit, reads as 1 if SVC call is active
[6]	-	Reserved, RES0.
[5]	NMIACT	NMI exception active state bit, reads as 1 if exception is active.
[4]	SECUREFAULTACT	RES0
[3]	USGFAULTACT	UsageFault exception active bit, reads as 1 if exception is active
[2]	HARDFAULTACT	HardFault exception active bit, reads as 1 if exception is active
[1]	BUSFAULTACT	BusFault exception active bit, reads as 1 if exception is active
[0]	MEMFAULTACT	MemManage exception active bit, reads as 1 if exception is active

Table 4-47 SHCSR bit assignments with the Security Extension

Bits	Name	Function
	-	Reserved, RES0.
[21]	HARDFAULTPENDED	HardFault exception pended state bit, set to 1 to allow exception modification. This bit is banked between Security states. **Note** The Non-secure HardFault exception does not preempt if AIRCR.BFHFNMINS is set to zero.[n]
[20]	SECUREFAULTPENDED	SecureFault exception pended state bit, set to 1 to allow exception modification. This bit is not banked between Security states. RES0 n
[19]	SECUREFAULTENA	SecureFault exception enable bit, set to 1 to enable. This bit is not banked between Security states. RES0
[18]	USGFAULTENA	UsageFault enable bit, set to 1 to enable.[m] This bit is banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-47 SHCSR bit assignments with the Security Extension (continued)

Bits	Name	Function
[17]	BUSFAULTENA	BusFault enable bit, set to 1 to enable.[m] If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state. This bit is not banked between Security states.
[16]	MEMFAULTENA	MemManage enable bit, set to 1 to enable. [m] This bit is banked between Security states.
[15]	SVCALLPENDED	SVCall pending bit, reads as 1 if exception is pending. [n] This bit is banked between Security states.
[14]	BUSFAULTPENDED	BusFault exception pending bit, reads as 1 if exception is pending. [n] If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state. This bit is not banked between Security states.
[13]	MEMFAULTPENDED	MemManage exception pending bit, reads as 1 if exception is pending.[n] This bit is banked between Security states.
[12]	USGFAULTPENDED	UsageFault exception pending bit, reads as 1 if exception is pending.[n] This bit is banked between Security states.
[11]	SYSTICKACT	SysTick exception active bit, reads as 1 if exception is active.[o] This bit is banked between Security states.
[10]	PENDSVACT	PendSV exception active bit, reads as 1 if exception is active. This bit is banked between Security states.
[9]	-	Reserved, RES0.
[8]	MONITORACT	Debug monitor active bit, reads as 1 if Debug monitor is active. This bit is not banked between Security states.
[7]	SVCALLACT	SVCall active bit, reads as 1 if SVC call is active. This bit is banked between Security states.
[6]	-	Reserved, RES0.
[5]	NMIACT	NMI exception active state bit, reads as 1 if exception is active. This bit is not banked between Security states.
[4]	SECUREFAULTACT	SecureFault exception active state bit, reads as 1 if exception is active. This bit is not banked between Security states.
[3]	USGFAULTACT	UsageFault exception active bit, reads as 1 if exception is active. This bit is banked between Security states.
[2]	HARDFAULTACT	HardFault exception active bit, reads as 1 if exception is active. This bit is banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-47 SHCSR bit assignments with the Security Extension (continued)

Bits	Name	Function
[1]	BUSFAULTACT	BusFault exception active bit, reads as 1 if exception is active. If AIRCR.BFHFNMINS is zero this bit is RAZ/WI from Non-secure state. This bit is not banked between Security states.
[0]	MEMFAULTACT	MemManage exception active bit, reads as 1 if exception is active. This bit is banked between Security states.

If you disable a system handler and the corresponding fault occurs, the processor treats the fault as a hard fault.

You can write to this register to change the pending or active status of system exceptions. An OS kernel can write to the active bits to perform a context switch that changes the current exception type.

Caution

- Software that changes the value of an active bit in this register without correct adjustment to the stacked content can cause the processor to generate a fault exception. Ensure software that writes to this register retains and restores the current active status.
- After you have enabled the system handlers, if you have to change the value of a bit in this register you must use a read-modify-write procedure. Using a read-modify-write procedure ensures that you change only the required bit.

4.3.33 System Handler Priority Registers

The SHPR1-SHPR3 registers set the priority level, 0 to 255 of the exception handlers that have configurable priority. SHPR1-SHPR3 are byte accessible.

See *4.3.1 System control block registers summary* on page 4-474 for the SHPR1-SHPR3 attributes.

These registers are banked between Security states on a bit field by bit field basis.

In an implementation with the Security Extension, These registers are banked between Security states on a bit field by bit field basis.

The system fault handlers and the priority field and register for each handler are:

Table 4-48 System fault handler priority fields

Handler	Field	Register description
MemManage	PRI_4	*System Handler Priority Register 1* on page 4-525
BusFault	PRI_5	
UsageFault	PRI_6	
SecureFault	PRI_7	
SVCall	PRI_11	*System Handler Priority Register 2* on page 4-525

[m] Enable bits, set to 1 to enable the exception, or set to 0 to disable the exception.
[n] Pending bits, read as 1 if the exception is pending, or as 0 if it is not pending. You can write to these bits to change the pending status of the exceptions.
[o] Active bits, read as 1 if the exception is active, or as 0 if it is not active. You can write to these bits to change the active status of the exceptions, but see the Caution in this section.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-48 System fault handler priority fields (continued)

Handler	Field	Register description
DebugMonitor	PRI_12	*System Handler Priority Register 3* on page 4-526
PendSV	PRI_14	
SysTick	PRI_15	

Each PRI_*n* field is 8 bits wide, but the processor implements only bits[7:M] of each field, and bits[M-1:0] read as zero and ignore writes, where M is the maximum supported priority number. Higher priority field values correspond to lower exception priorities.

System Handler Priority Register 1

SHPR1 sets or returns priority for system handlers 4-7.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED18`.
Non-secure alias is provided using SHPR1_NS, located at `0xE002ED18`.
A 32-bit RW register located at `0xE000ED18`. Non-secure alias is provided using SHPR1_NS, located at `0xE002ED18`.
This register is banked between Security states on a bit-by-bit basis.

The following figure shows the SHPR1 bit assignments.

31 24	23 16	15 8	7 0
PRI_7	PRI_6	PRI_5	PRI_4

Figure 4-27 SHPR1 bit assignments

The following table describes the SHPR1 bit assignments.

Table 4-49 SHPR1 register bit assignments

Bits	Name	Function	Security state
[31:24]	PRI_7	Priority of system handler 7, SecureFault Always RAZ/WI	PRI_7 is RAZ/WI from Non-secure state.
[23:16]	PRI_6	Priority of system handler 6, UsageFault	PRI_6 is banked between Security states.
[15:8]	PRI_5	Priority of system handler 5, BusFault	PRI_5 is RAZ/WI from Non-secure state if AIRCR.BFHFNMINS is 0.
[7:0]	PRI_4	Priority of system handler 4, MemManage	PRI_4 is banked between Security states.

System Handler Priority Register 2

SHPR2 sets or returns priority for system handlers 8-11.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
A 32-bit RW register located at `0xE000ED1C`.
Non-secure alias is provided using SHPR2_NS, located at `0xE002ED1C`.
A 32-bit RW register located at `0xE000ED1C`. Non-secure alias is provided using SHPR2_NS, located at `0xE002ED1C`.
This register is banked between Security states.

The following figure shows the SHPR2 bit assignments.

31 24	23 0
PRI_11	Reserved

Figure 4-28 SHPR2 bit assignments

The following table describes the SHPR2 bit assignments.

Table 4-50 SHPR2 register bit assignments

Bits	Name	Function
[31:24]	PRI_11	Priority of system handler 11, SVCall
[23:0]	-	Reserved, RES0.

System Handler Priority Register 3

SHPR3 sets or returns priority for system handlers 12-15.

Usage constraints
Privileged access permitted only. Unprivileged accesses generate a fault.

Configuration
This register is always implemented.

Attributes
A 32-bit RW register located at `0xE000ED20`.
Non-secure alias is provided using SHPR3_NS, located at `0xE002ED20`.
A 32-bit RW register located at `0xE000ED20`. Non-secure alias is provided using SHPR3_NS, located at `0xE002ED20`.
This register is banked between Security states on a bit-by-bit basis.

The following figure shows the SHPR3 bit assignments.

31 24	23 16	15 0
PRI_15	PRI_14	Reserved

Figure 4-29 SHPR3 bit assignments

The following table describes the SHPR3 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-51 SHPR3 register bit assignments

Bits	Name	Function	Security state
[31:24]	PRI_15	Priority of system handler 15, SysTick exception	PRI_15 is banked between Security states.
[23:16]	PRI_14	Priority of system handler 14, PendSV	PRI_14 is is banked between Security states.
[15:8]	-	Reserved, RES0.	-
[7:0]	PRI_12	Priority of system handler 12, DebugMonitor.	PRI_14 is is not banked between Security states.

4.3.34 Revision ID Register, REVIDR

The REVIDR register provides additional implementation-specific minor revision that can be interpreted with the CPUID register.

Usage constraints

Unprivileged access results in a BusFault exception.
The Non-secure version of this register, REVIDR_NS is located at `0xE002ECFC`.
If the Security Extension is implemented, the Non-secure version of this register, REVIDR_NS is located at `0xE002ECFC`.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the REVIDR bit assignments.

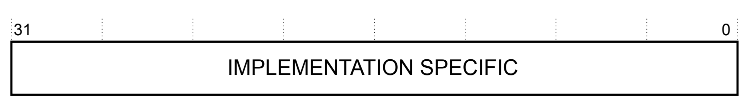


Figure 4-30 REVIDR bit assignments

The following table describes the REVIDR bit assignments.

Table 4-52 REVIDR bit assignments

Field	Name	Type	Description
[31:0]	IMPLEMENTATION SPECIFIC	RO	Implementation-specific minor revision information that can be interpreted with the CPUID register. <Add the relevant minor revision information specific to your implementation of Cortex-M55.>

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

For more information on the CPUID register, see the *Arm®v8-M Architecture Reference Manual*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.35 Vector Table Offset Register

The VTOR indicates the offset of the vector table base address from memory address `0x00000000`.

Usage constraints

See *4.3.1 System control block registers summary* on page 4-474 for the VTOR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000ED08`. When the Security Extension is implemented, the Non-secure alias is provided using VTOR_NS, located at `0xE002ED08`.
A 32-bit RW register located at `0xE000ED08`.
The Non-secure alias is provided using VTOR_NS, located at `0xE002ED08`
This register is banked between Security states.

In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the VTOR bit assignments.

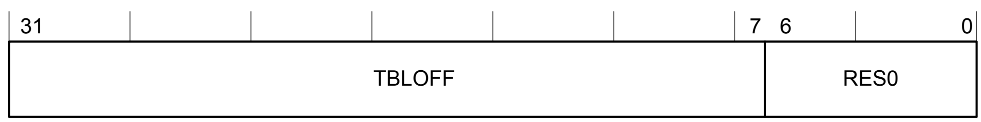


Figure 4-31 VTOR bit assignments

The following table describes the VTOR bit assignments.

Table 4-53 VTOR bit assignments

Bits	Name	Function
[31:7]	TBLOFF	Vector table base offset field. Bits [31:7] of VTOR_S are based on bits [31:7] of the Secure vector table initialization signal. Bits [31:7] of VTOR_NS are based on bits [31:7] of the Non-secure vector table initialization signal. Bits [31:7] of VTOR are based on bits [31:7] of the vector table initialization signal. <See the configurable information after this table for information about the configuration of this field and the [6:0] field that follows.>.
[6:0]	-	Reserved, RES0.

When setting TBLOFF, you must align the offset to the number of exception entries in the vector table. <Configure the next statement to give the information required for your implementation, the statement reminds you of how to determine the alignment requirement.> The minimum alignment is 32 words, enough for up to 16 interrupts. For more interrupts, adjust the alignment by rounding up to the next power of two. For example, if you require 21 interrupts, the alignment must be on a 64-word boundary because the required table size is 37 words, and the next power of two is 64.

Table alignment requirements mean that bits[6:0] of the table offset are always zero.

Note

The processor has external signals that disable writes to VTOR_S and VTOR_NS from software or from a debug agent that is connected to the processor.

The processor has external signals that disable writes to VTOR from software or from a debug agent that is connected to the processor.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.3.36 System Control Block design hints and tips

Ensure software uses aligned accesses of the correct size to access the system control block registers:

- Except for the CFSR and SHPR1-SHPR3, it must use aligned word accesses.
- For the CFSR and SHPR1-SHPR3 it can use byte or aligned halfword or word accesses.

In a fault handler, to determine the true faulting address:

1. Read and save the MMFAR or BFAR value.
2. Read the MMARVALID bit in the MMFSR, or the BFARVALID bit in the BFSR. The MMFAR or BFAR address is valid only if this bit is 1.

Software must follow this sequence because another higher priority exception might change the MMFAR or BFAR value. For example, if a higher priority handler pre-empts the current fault handler, the other fault might change the MMFAR or BFAR value.

4.3.37 Implementation control block register summary

The following table shows the ICB registers that provide system implementation-specific information

Table 4-54 ICB register summary

Address	Name	Type	Required privilege	Reset value	Description
`0xE000E004`	ICTR	RO	Privileged	`0x0000000X` **Note** ICTR[3:0] depends on the number of interrupts included in the processor. Bits [31:4] are zero.	*4.3.39 Interrupt Controller Type Register* on page 4-532
`0xE000E008`	ACTLR	RW	Privileged	`0x00000000`	*4.3.38 Auxiliary Control Register* on page 4-530
`0xE000E00C`	CPPWR	RW	Privileged	`0x00000000`	*4.3.40 Coprocessor Power Control Register* on page 4-533

4.3.38 Auxiliary Control Register

The ACTLR contains a number of fields that allow software to control the processor features and functionality.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a BusFault.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000E008`.
Non-secure alias is provided using ACTLR_NS, located at `0xE002E008`.
If the Security Extension is implemented, Non-secure alias is provided using ACTLR_NS, located at `0xE002E008`.
This register is banked between Security states.
If the Security Extension is implemented, this register is banked between Security states.

The following figure shows the ACTLR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

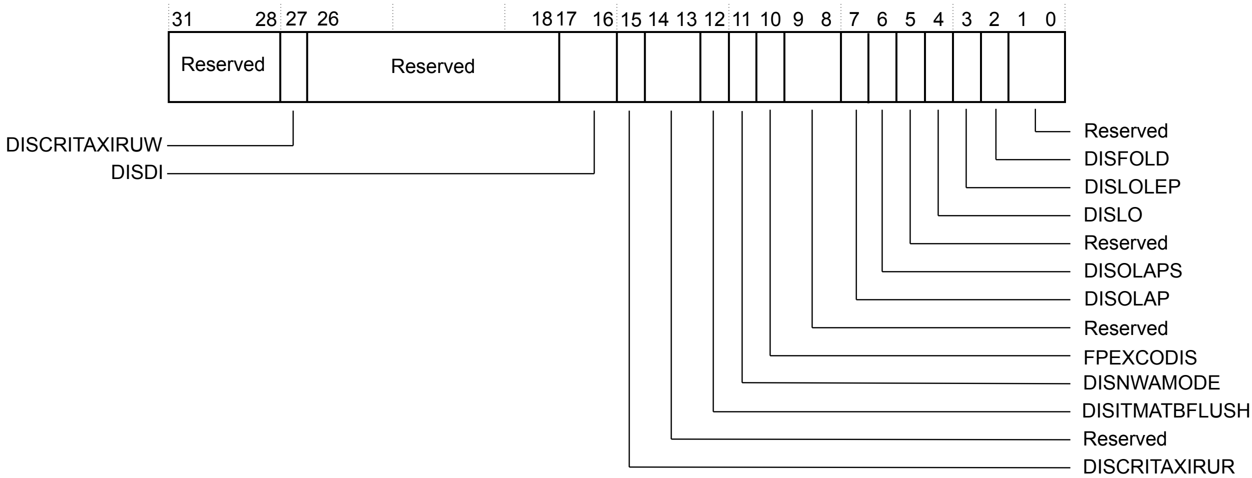


Figure 4-32 ACTLR bit assignments

The following table describes the ACTLR bit assignments.

Table 4-55 ACTLR bit assignments

Bits	Name	Type	Description
[31:28]	Reserved	-	These bits are reserved for future use and must be treated as UNK/SBZP.
[27]	DISCRITAXIRUW	RW	Disable-Critical-AXI-Read-Under-Write. The options are: **0** Normal operation. **1** AXI reads to Device memory and exclusive reads to shared memory are not initiated on the M-AXI read address channel until all outstanding writes on the M-AXI interface are complete. Setting this bit decreases performance.
[26:19]	Reserved	-	These bits are reserved for future use and must be treated as UNK/SBZP.
[18]	DISGSO	RW	Disable gather and scatter optimization for contiguous accesses. Setting this bit decreases performance.
[17:16]	DISDI	RW	Disable dual-issue features. The options for this bit are: `0b00` Full dual-issue, if DISFOLD is set to 0. `0b01` Disable dual-issue of arithmetic instructions. `0b10` Disable lane swapping `0b11` Disable dual-issue of arithmetic instructions and lane swapping.
[15]	DISCRITAXIRUR	RW	Disable critical AXI Read-Under-Read. The options for this bit are: **0** Normal operation. **1** AXI reads to Device memory and exclusive reads to shared memory are not initiated on the M-AXI read address channels if there are any outstanding reads on the M-AXI. Transactions on the M-AXI cannot be interrupted. This bit might reduce the time these transactions are in progress and might improve worst-case interrupt latency. Setting this bit reduces performance.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-55 ACTLR bit assignments (continued)

Bits	Name	Type	Description
[14:13]	Reserved	-	These bits are reserved for future use and must be treated as UNK/SBZP.
[12]	DISITMATBFLUSH	RW	This bit determines whether *Instrumentation Trace Macrocell* (ITM) or *Data Watchpoint and Trace* (DWT) ATB flush is disabled. The options for this bit are: **0** Normal operation. **1** ITM or DWT ATB flush is disabled. When disabled, the **AFVALID** signal (trace flush request) is ignored and the **AFREADY** (trace flush ready) signal is held HIGH.
[11]	DISNWAMODE	RW	This bit determines whether no write allocate mode is disabled. The options for this bit are: **0** Normal operation. **1** No write allocate mode is disabled. Setting this bit decreases performance.
[10]	FPEXCODIS	RW	This bit determines whether floating-point exception outputs are disabled. The options for this bit are: **0** Normal operation. **1** Floating-point exception outputs are disabled.
[9:8]	Reserved	-	These bits are reserved for future use and must be treated as UNK/SBZP.
[7]	DISOLAP	RW	Disable overlapping of all instructions.
[6]	DISOLAPS	RW	Disable overlapping of scalar-only instructions.
[5]	Reserved	-	UNK/SBZP
[4]	DISLO	RW	Disable low overhead loops. The options are: **0** Low overhead loops enabled. **1** Low overhead loops disabled.
[3]	Reserved	-	RES0
[2]	DISFOLD	RW	This bit determines whether dual-issue functionality is disabled. The options are: **0** Normal operation. **1** Dual-issue functionality is disabled. Setting this bit decreases performance.
[1]	Reserved	-	These bits are reserved for future use and must be treated as UNK/SBZP.
[0]	DISMCYCINT	RW	This bit can be set to disable interruption of multi-cycle instructions. The options are: **0** Normal operation. **1** Disables interruption of multi-cycle instructions. This increases the interrupt latency of the processor because Load/Store and Multiply/Divide operations complete before the interrupt stacking occurs.

4.3.39 Interrupt Controller Type Register

The ICTR provides information about the interrupt controller.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints

See *4.3.37 Implementation control block register summary* on page 4-530 for the ICTR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RO register located at `0xE000E004`.
Non-secure alias is provided using ICTR_NS, located at `0xE002E004`.
If the Security Extension is implemented, Non-secure alias is provided using ICTR_NS, located at `0xE002E004`.
This register is not banked between Security states.
If the Security Extension is implemented, this register is not banked between Security states.

The following figure shows the ICTR bit assignments.

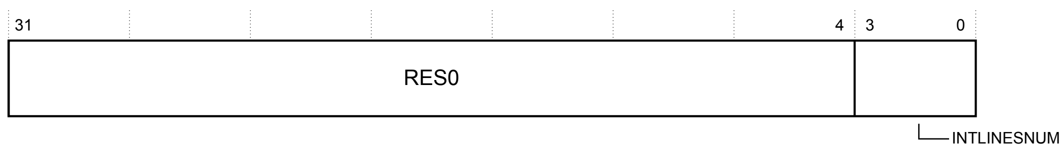


Figure 4-33 ICTR bit assignments

The following table describes the ICTR bit assignments.

Table 4-56 ICTR bit assignments

Bits	Name	Function
[31:4]	-	Reserved, RES0.
[3:0]	INTLINESNUM	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_IPRn, 4×INTLINESNUM. <See the configurable information after this table for information about the configuration of this field and the [6:0] field that follows.>.

<Configure the next statement to give the information required for your implementation, the statement reminds you of how to determine the interrupt line set number.> The allowed range of interrupts is 1-480, indicated by IRQ0-IRQ479.

4.3.40 Coprocessor Power Control Register

The CPPWR register specifies whether coprocessors are permitted to enter a non-retentive power state.

Usage constraints

See *4.3.37 Implementation control block register summary* on page 4-530 for the ICTR attributes.

Configuration

This register is always implemented.

Attributes

A 32-bit RW register located at `0xE000E00C`.
Non-secure alias is provided using CPPWR_NS, located at `0xE002E00C`.
If the Security Extension is implemented, Non-secure alias is provided using CPPWR_NS, located at `0xE002E00C`.
This register is not banked between Security states.
If the Security Extension is implemented, this register is not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following figure shows the CPPWR bit assignments.

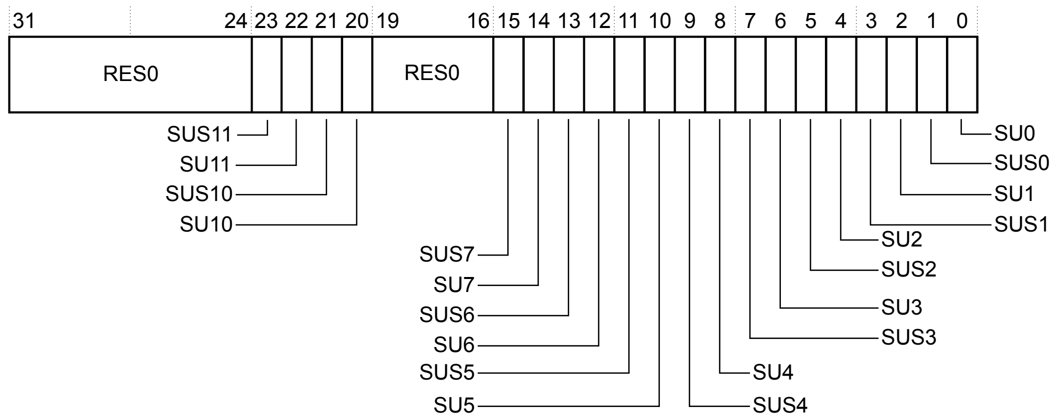


Figure 4-34 CPPWR bit assignments

The following table shows the CPPWR bit assignments.

Table 4-57 CPPWR bit assignments

Bits	Name	Function
[31:24]	-	Reserved, RES0
[23]	SUS11	State UNKNOWN Secure only 11. The value in this field is ignored. If this Security Extension is implemented, this bit denotes state UNKNOWN Secure only 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SUS10 field, then the value is UNKNOWN. This bit is RAZ/WI from Non-secure state. If the Security Extension is implemented, this bit is RAZ/WI from Non-secure state.
[22]	SU11	State UNKNOWN 11. The value in this field is ignored. If the value of this bit is not programmed to the same value as the SU10 field, then the value is UNKNOWN. When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. If the Security Extension is implemented and when SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-57 CPPWR bit assignments (continued)

Bits	Name	Function
[21]	SUS10	State UNKNOWN Secure only 10. This bit indicates and allows modification of whether the SU10 field can be modified from Non-secure state. If this Security Extension is implemented, this bit denotes 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: `0b0` The SU10 field is accessible from both Security states. `0b1` The SU10 field is only accessible from the Secure state. If the Security Extension is implemented, the possible values of this bit are: `0b0` The SU10 field is accessible from both Security states. `0b1` The SU10 field is only accessible from the Secure state. This bit is RAZ/WI from Non-secure state. If Security Extension is implemented, this bit is RAZ/WI from Non-secure state.
[20]	SU10	State UNKNOWN 10. This bit indicates and allows modification of whether the state associated with the floating-point and *M-profile Vector Extension* (MVE) functionality is permitted to become UNKNOWN. This can be used as a hint to power control logic that the *Extension Processing Unit* (EPU) might be powered down. **0** The floating-point and *M-profile Vector Extension* (MVE) functionality is not permitted to become UNKNOWN. **1** The floating-point and *M-profile Vector Extension* (MVE) functionality is permitted to become UNKNOWN. When SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. If the Security Extension is implemented and when SUS10 is set to 1, the Non-secure view of this bit is RAZ/WI. If Security Extension is implemented, this bit is RAZ/WI from Non-secure state.
[19:16]	-	Reserved, RES0

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-57 CPPWR bit assignments (continued)

Bits	Name	Function
SUS*m*, bit [2*m*+1:2*m*], for *m* = 0-7	SUS*m*	State UNKNOWN Secure only *m*.This field indicates and allows modification of whether the SU*m* field can be modified from Non-secure state. If the Security Extension is implemented, this bit indicates that the state is 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 bit are: **0** The SU*m* field is accessible from both Security states. **1** The SU*m* field is only accessible from the Secure state. **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.
SU*m*, bit [2*m*], for *m* = 0-7	SU*m*	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. **0** The coprocessor state is not permitted to become UNKNOWN. **1** The coprocessor state is permitted to become UNKNOWN. When SU*m* is set to 1, the Non-secure view of this bit is RAZ/WI. If Security Extension is implemented and when SU*m* is set to 1, the Non-secure view of this bit is RAZ/WI.

4.3.41 NVIC usage hints and tips

Ensure that software uses correctly aligned register accesses. The processor does not support unaligned accesses to NVIC registers.

An interrupt can enter pending state even if it is disabled. Disabling an interrupt only prevents the processor from taking that interrupt.

Before programming VTOR to relocate the vector table, ensure that the vector table entries of the new vector table are set up for fault handlers, NMI, and all enabled exceptions like interrupts.

NVIC programming hints

Software uses the `CPSIE i` and `CPSID i` instructions to enable and disable interrupts.

The CMSIS provides the following intrinsic functions for these instructions:

```
void __disable_irq(void) // Disable Interrupts
void __enable_irq(void)  // Enable Interrupts
```

In addition, the CMSIS provides functions for NVIC control, listed in *4.2.1 Accessing the NVIC registers using CMSIS* on page 4-466.

The input parameter `IRQn` is the IRQ number, see *2.3.2 Exception types* on page 2-59 for more information. For more information about these functions, see the CMSIS documentation.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.4 System timer, SysTick

In a implementation with Security Extension, there are two 24-bit system timers, a Non-secure SysTick timer and a Secure SysTick timer. In an implementation without the Security Extension, only a single a 24-bit system timer, SysTick is used.

When enabled, the timer counts down from the reload value to zero, reloads (wraps to) the value in the SYST_RVR on the next clock cycle, then decrements on subsequent clock cycles. Writing a value of zero to the SYST_RVR disables the counter on the next wrap. When the counter transitions to zero, the COUNTFLAG status bit is set to 1. Reading SYST_CSR clears the COUNTFLAG bit to 0. Writing to the SYST_CVR clears the register and the COUNTFLAG status bit to 0. The write does not trigger the SysTick exception logic. Reading the register returns its value at the time it is accessed.

Note

When the processor is halted for debugging, the counter does not decrement.

The system timer registers are:

Table 4-58 System timer registers summary

Address	Name	Type	Reset value	Description
`0xE000E010`	SYST_CSR	RW	`0x00000000`	*4.4.1 SysTick Control and Status Register* on page 4-537.
`0xE000E014`	SYST_RVR	RW	UNKNOWN	*4.4.2 SysTick Reload Value Register* on page 4-538.
`0xE000E018`	SYST_CVR	RW	UNKNOWN	*4.4.3 SysTick Current Value Register* on page 4-538.
`0xE000E01C`	SYST_CALIB	RO	`0xC0000000` (SysTick calibration value)	*4.4.4 SysTick Calibration Value Register* on page 4-539.

4.4.1 SysTick Control and Status Register

The SYST_CSR controls and provides status date for the SysTick timer.

In an implementation with the Security Extension, this register is banked between Security states.

The bit assignments for SYST_CSR are:

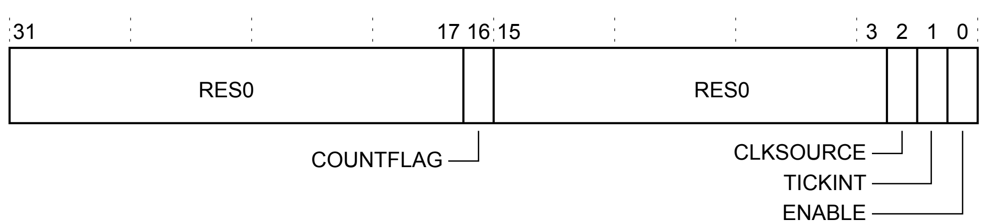


Table 4-59 SYST_CSR bit assignments

Bits	Name	Function
[31:17]	-	Reserved, RES0.
[16]	COUNTFLAG	Returns 1 if timer counted to 0 since the last read of this register.
[15:3]	-	Reserved, RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-59 SYST_CSR bit assignments (continued)

Bits	Name	Function
[2]	CLKSOURCE	Selects the SysTick timer clock source: **0** External reference clock. **1** Processor clock.
[1]	TICKINT	Enables SysTick exception request: **0** Counting down to zero does not assert the SysTick exception request. **1** Counting down to zero asserts the SysTick exception request.
[0]	ENABLE	Enables the counter: **0** Counter disabled. **1** Counter enabled.

4.4.2 SysTick Reload Value Register

The SYST_RVR specifies the SysTick timer counter reload value.

See *4.4 System timer, SysTick* on page 4-537 for the SYST_RVR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The bit assignments for SYST_RVR are:

31 – 24	23 – 0
RES0	RELOAD

Table 4-60 SYST_RVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved, RES0.
[23:0]	RELOAD	Value to load into the SYST_CVR when the counter is enabled and when it reaches 0, see *Calculating the RELOAD value* on page 4-538.

Calculating the RELOAD value

The SYST_RVR specifies the SysTick timer counter reload value.

The RELOAD value can be any value in the range `0x00000001-0x00FFFFFF`. You can program a value of 0, but this has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

To generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. For example, if the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.

4.4.3 SysTick Current Value Register

The SYST_CVR contains the current value of the SysTick counter.

See *4.4 System timer, SysTick* on page 4-537 for the SYST_CVR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The bit assignments for SYST_CVR:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

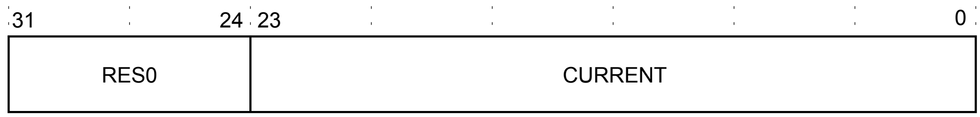


Table 4-61 SYST_CVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved, RES0.
[23:0]	CURRENT	Reads the current value of the SysTick counter. A write of any value clears the field to 0, and also clears the SYST_CSR.COUNTFLAG bit to 0.

4.4.4 SysTick Calibration Value Register

The SYST_CALIB register indicates the SysTick calibration value and parameters for the selected Security state.

See *4.4 System timer, SysTick* on page 4-537 for the SYST_CALIB attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The bit assignments for SYST_CALIB_S and SYST_CALIB_S are:

The following figure shows the bit assignments for SYST_CALIB. If the Security Extension is implemented, then the bit assignment applies to SYST_CALIB_S and SYST_CALIB_NS.

The bit assignments for SYST_CALIB_S are:

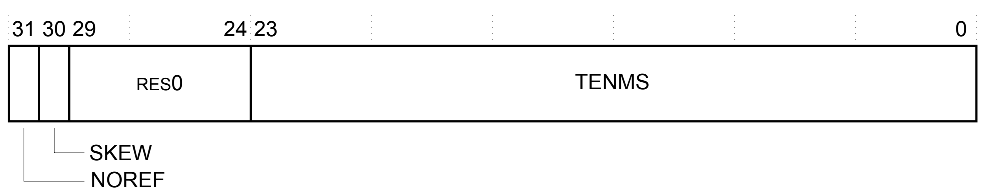


Figure 4-35 SYST_CALIB_S bit assignments

Table 4-62 SYST_CALIB_S bit assignments

Bits	Name	Function
[31]	NOREF	Indicates whether the device provides a reference clock to the processor: **0** Reference clock provided. **1** No reference clock provided. If your device does not provide a reference clock, the SYST_CSR.CLKSOURCE bit reads-as-one and ignores writes.
[30]	SKEW	Indicates whether the TENMS value is exact: **0** TENMS value is exact. **1** TENMS value is inexact, and has a rounding error of +/- 0.5 of one LSB. An inexact TENMS value can affect the suitability of SysTick as a software real time clock.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-62 SYST_CALIB_S bit assignments (continued)

Bits	Name	Function
[29:24]	-	Reserved.
[23:0]	TENMS	Reload value for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as zero, the calibration value is not known.

The bit assignments for SYST_CALIB_NS are:

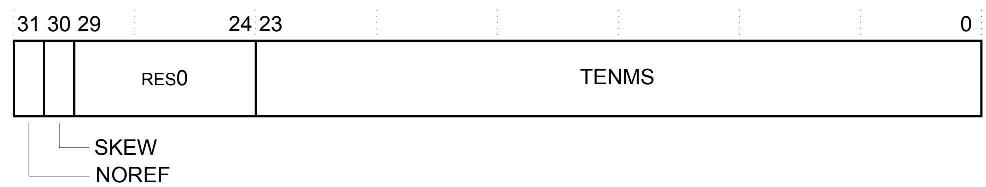


Figure 4-36 SYST_CALIB_NS bit assignments

Table 4-63 SYST_CALIB_NS bit assignments

Bits	Name	Function
[31]	NOREF	Indicates whether the device provides a reference clock to the processor: **0** Reference clock provided. **1** No reference clock provided. If your device does not provide a reference clock, the SYST_CSR.CLKSOURCE bit reads-as-one and ignores writes.
[30]	SKEW	Indicates whether the TENMS value is exact: **0** TENMS value is exact. **1** TENMS value is inexact, and has a rounding error of +/- 0.5 of one LSB. An inexact TENMS value can affect the suitability of SysTick as a software real time clock.
[29:24]	-	Reserved.
[23:0]	TENMS	Reload value for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as zero, the calibration value is not known.

The bit assignments for SYST_CALIB are:

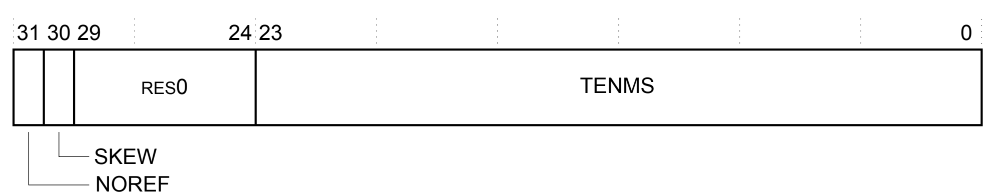


Figure 4-37 SYST_CALIB bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-64 SYST_CALIB bit assignments

Bits	Name	Function
[31]	NOREF	Indicates whether the device provides a reference clock to the processor: **0** Reference clock provided. **1** No reference clock provided. If your device does not provide a reference clock, the SYST_CSR.CLKSOURCE bit reads-as-one and ignores writes.
[30]	SKEW	Indicates whether the TENMS value is exact: **0** TENMS value is exact. **1** TENMS value is inexact, and has a rounding error of +/- 0.5 of one LSB. An inexact TENMS value can affect the suitability of SysTick as a software real time clock.
[29:24]	-	Reserved.
[23:0]	TENMS	Reload value for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as zero, the calibration value is not known.

Table 4-65 SYST_CALIB register bit assignments

Bits	Name	Function
[31]	NOREF	**0** Reference clock is implemented, and implies which clock source is described by SKEW and TENMS. **1** Indicates that no separate reference clock is provided.
[30]	SKEW	**0** Calibration value is exact. **1** Reads as one. Calibration value for the 10ms inexact timing is not known because TENMS is not known. This can affect the suitability of SysTick as a software real-time clock.
[29:24]	-	Reserved, RES0.
[23:0]	TENMS	Reads as zero. Indicates that the calibration value is not known. Optionally holds a reload value to be used for 10ms (100Hz) timing. If NOREF=0, TENMS is the reload value to generate 10ms using the external reference clock. If NOREF=1, TENMS holds the reload value to generate 10ms using the processor clock. If the provided clocks do not allow a consistent 10ms timing period, for example, because the clock speeds change or the SysTick is clock gated or powered down, TENMS should be zero to indicate this.

If calibration information is not known, calculate the calibration value required from the frequency of the core clock or external clock.

4.4.5 SysTick usage hints and tips

The interrupt controller clock updates the SysTick counter. If this clock signal is stopped for low-power mode, the SysTick counter stops.

Ensure software uses word accesses to access the SysTick registers.

If the SysTick counter reload and current value are undefined at reset, the correct initialization sequence for the SysTick counter is:
1. Program reload value.
2. Clear current value.
3. Program SYST_CSR.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.5 Cache maintenance operations

The cache maintenance operation registers control the data and instruction cache.

The operations supported for the instruction and data cache are:

- Enabling and disabling the cache.
- Invalidating the cache.
- Cleaning the cache.

The cache maintenance operations are only accessible by privileged loads and stores. Unprivileged accesses to these registers always generate a BusFault.

The following table lists the cache maintenance operation registers.

Table 4-66 Cache maintenance register summary

Address	Name	Type	Privilege	Reset value	Description
`0xE000EF50`	ICIALLU	WO	Privileged	UNKNOWN	*4.5.1 Instruction Cache Invalidate All to PoU, ICIALLU* on page 4-543
`0xE000EF54`	-	-	-	-	Reserved
`0xE000EF58`	ICIMVAU	WO	Privileged	UNKNOWN	*4.5.2 Instruction Cache line Invalidate by Address to PoU, ICIMVAU* on page 4-543
`0xE000EF5C`	DCIMVAC	WO	Privileged	UNKNOWN	*4.5.3 Data Cache line Invalidate by Address to PoC, DCIMVAC* on page 4-544
`0xE000EF60`	DCISW	WO	Privileged	UNKNOWN	*4.5.4 Data Cache line Invalidate by Set/Way, DCISW* on page 4-545
`0xE000EF64`	DCCMVAU	WO	Privileged	UNKNOWN	*4.5.5 Data cache clean by address to the PoU, DCCMVAU* on page 4-546
`0xE000EF68`	DCCMVAC	WO	Privileged	UNKNOWN	*4.5.8 Data Cache Clean and Invalidate by Address to the PoC, DCCIMVAC* on page 4-548
`0xE000EF6C`	DCCSW	WO	Privileged	UNKNOWN	*4.5.7 Data Cache Clean line by Set/Way, DCCSW* on page 4-547
`0xE000EF70`	DCCIMVAC	WO	Privileged	UNKNOWN	*4.5.8 Data Cache Clean and Invalidate by Address to the PoC, DCCIMVAC* on page 4-548
`0xE000EF74`	DCCISW	WO	Privileged	UNKNOWN	*4.5.9 Data Cache Clean and Invalidate by Set/Way, DCCISW* on page 4-549
`0xE000EF78`	BPIALL	WO	Privileged	UNKNOWN	*4.5.10 Branch Predictor Invalidate All, BPIALL* on page 4-550

This section contains the following subsections:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.5.1 Instruction Cache Invalidate All to PoU, ICIALLU

The ICIALLU invalidates all instruction caches to *Point of Unification* (PoU).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the ICIALLU bit assignments.

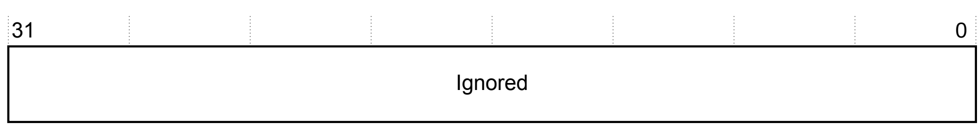


Figure 4-38 ICIALLU bit assignments

The following table describes the ICIALLU bit assignments.

Table 4-67 ICIALLU bit assignments

Bits	Name	Function
[31:0]	Ignored	The value written to this field is ignored.

4.5.2 Instruction Cache line Invalidate by Address to PoU, ICIMVAU

The ICIMVAU invalidates instruction cache lines by address to *Point of Unification* (PoU).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the ICIMVAU bit assignments.

Figure 4-39 ICIMVAU bit assignments

The following table describes the ICIMVAU bit assignments.

Table 4-68 ICIMVAU bit assignments

Bits	Name	Function
[31:0]	Address	Writing to this field initiates the maintenance operation for the address that is written

4.5.3 Data Cache line Invalidate by Address to PoC, DCIMVAC

The DCIMVAC invalidates data or unified cache lines by address to *Point of Coherency* (PoC).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCIMVAC bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Figure 4-40 DCIMVAC bit assignments

The following table describes the DCIMVAC bit assignments.

Table 4-69 DCIMVAC bit assignments

Bits	Name	Function
[31:0]	Address	Writing to this field initiates the maintenance operation for the address that is written

4.5.4 Data Cache line Invalidate by Set/Way, DCISW

The DCISW invalidates instruction cache lines by address to *Point of Unification* (PoU).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCISW bit assignments.

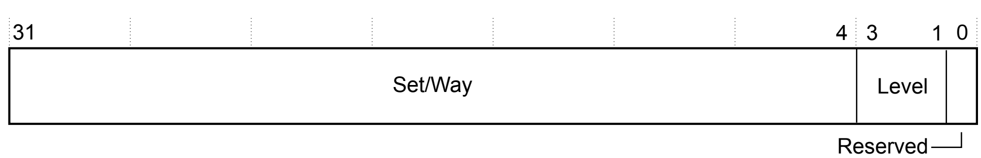


Figure 4-41 DCISW bit assignments

The following table describes the DCISW bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-70 DCISW bit assignments

Bits	Name	Function
[31:4]	SetWay	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 = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(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.
[3:1]	Level	Cache level. Cache level to operate on, minus 1.
[0]	-	Reserved, RES0.

4.5.5 Data cache clean by address to the PoU, DCCMVAU

The DCCMVAU cleans data or unified cache lines by address to *Point of Unification* (PoU).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCCMVAU bit assignments.

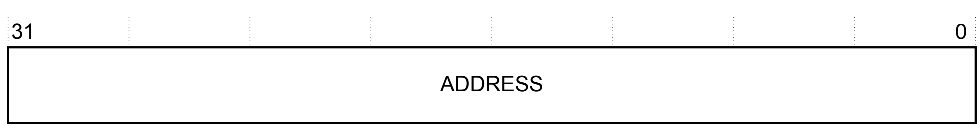


Figure 4-42 DCCMVAU bit assignments

The following table describes the DCCMVAU bit assignments.

Table 4-71 DCCMVAU bit assignments

Bits	Name	Function
[31:0]	Address	Writing to this field initiates the maintenance operation for the address that is written.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.5.6 Data cache line clean by address to the PoC, DCCMVAC

The DCCMVAC cleans data or unified cache lines by address to *Point of Unification* (PoC).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCCMVAC bit assignments.

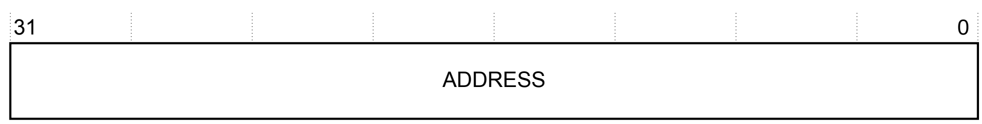


Figure 4-43 DCCMVAC bit assignments

The following table describes the DCCMVAC bit assignments.

Table 4-72 DCCMVAC bit assignments

Bits	Name	Function
[31:0]	Address	Writing to this field initiates the maintenance operation for the address that is written.

4.5.7 Data Cache Clean line by Set/Way, DCCSW

The DCISW cleans data or unified cache line by set/way.

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

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.

If the Security Extension is included, 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.

The following figure shows the DCCSW bit assignments.

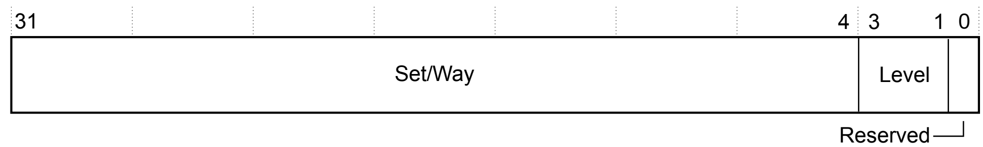


Figure 4-44 DCCSW bit assignments

The following table describes the DCCSW bit assignments.

Table 4-73 DCCSW bit assignments

Bits	Name	Function
[31:4]	SetWay	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 = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(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.
[3:1]	Level	Cache level. Cache level to operate on, minus 1.
[0]	-	Reserved, RES0.

4.5.8 Data Cache Clean and Invalidate by Address to the PoC, DCCIMVAC

The DCCIMVAC cleans data or unified cache lines by address to *Point of Coherency* (PoC).

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCCIMVAC bit assignments.

Figure 4-45 DCCIMVAC bit assignments

The following table describes the DCCIMVAC bit assignments.

Table 4-74 DCCIMVAC bit assignments

Bits	Name	Function
[31:0]	Address	Writing to this field initiates the maintenance operation for the address that is written.

4.5.9 Data Cache Clean and Invalidate by Set/Way, DCCISW

The DCISW cleans and invalidates data or unified cache line by set/way.

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the DCCISW bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

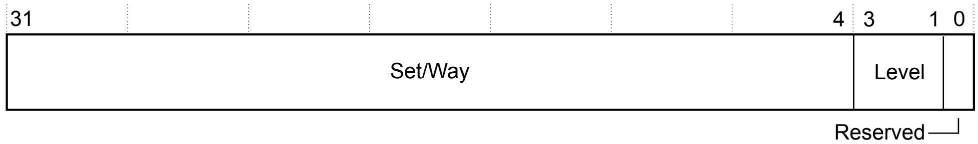


Figure 4-46 DCCISW bit assignments

The following table describes the DCCISW bit assignments.

Table 4-75 DCCISW bit assignments

Bits	Name	Function
[31:4]	SetWay	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 = Log2(ASSOCIATIVITY), L = Log2(LINELEN), B = (L + S), S = Log2(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.
[3:1]	Level	Cache level. Cache level to operate on, minus 1.
[0]	-	Reserved, RES0.

4.5.10 Branch Predictor Invalidate All, BPIALL

The BPIALL invalidates all entries from branch predictors.

Usage constraints

Privileged access only. Unprivileged access generates a fault.

This register is word accessible only. Halfword and byte accesses are UNPREDICTABLE.

This register is accessible to accesses through unprivileged DAP requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is always implemented.

Attributes

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.
If the Security Extension is included, 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.

The following figure shows the BPIALL bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

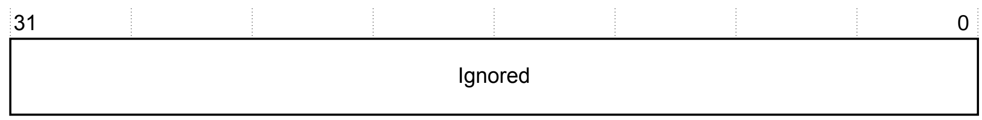


Figure 4-47 BPIALL bit assignments

The following table describes the BPIALL bit assignments.

Table 4-76 BPIALL bit assignments

Bits	Name	Function
[31:0]	Ignored	The value written to this field is ignored.

4.5.11 Accessing the NVIC cache maintenance operations using CMSIS

CMSIS functions enable software portability between different Cortex-M profile processors.

To access cache maintenance operations when using CMSIS, use the following functions:

Table 4-77 CMSIS access cache maintenance operations

CMSIS function	Description
`void SCB_EnableICache (void)`	Invalidate and then enable instruction cache
`void SCB_DisableICache (void)`	Disable instruction cache and invalidate its contents
`void SCB_InvalidateICache (void)`	Invalidate instruction cache
`void SCB_EnableDCache (void)`	Invalidate and then enable data cache
`void SCB_DisableDCache (void)`	Disable data cache and then clean and invalidate its contents
`void SCB_InvalidateDCache (void)`	Invalidate data cache
`void SCB_CleanDCache (void)`	Clean data cache
`void SCB_CleanInvalidateDCache (void)`	Clean and invalidate data cache
`void SCB_InvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Invalidate data cache by address
`void SCB_CleanDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Clean data cache by address
`void SCB_CleanInvalidateDCache_by_Addr (uint32_t *addr, int32_t dsize)`	Clean and invalidate data cache by address

Arm might add more cache management functions to the CMSIS in the future, and recommends that you check the CMSIS documentation on a regular basis for the latest information.

4.5.12 Initializing the instruction and data cache

On initial powerup, the instruction and data caches are in an UNKNOWN state. Therefore, on initial powerup, the caches must be initialized either by automatic invalidation or through software invalidation. Subsequently, under normal circumstances, the top-level **INITL1RSTDIS** signal is tied LOW.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Therefore, the instruction and data caches are automatically invalidated on powerup and no software invalidation is required.

If you implement RAM retention without using the P-Channel, then software invalidation of caches might be required.

If a P-Channel is used for RAM retention, then automatic invalidation is not required after returning from a state of retention because the cache content is already valid. In this case, the **INITL1RSTDIS** signal is expected to be LOW to carry out the initial invalidation, and subsequently, prevents automatic invalidation and cache invalidation because the cache content is already valid.

The caches are not accessible during the automatic invalidation sequence. Executing a `DSB` instruction causes the processor to wait for the sequence to complete.

The CCR.DC and CCR.IC register bits are banked based on security, therefore each Security state must set these bits to enable the data and instruction cache.

For more information on the CCR register, see *Arm®v8-M Architecture Reference Manual*.

Note

You can optionally implement *Error Correcting Code* (ECC) functionality on caches by setting the `ECC` RTL parameter. However, the Cortex-M55 processor does not support disabling ECC using software. Enabling and disabling ECC is done at Cold reset by the **INITECCEN** signal.

4.5.13 Enabling the instruction and data cache

The following code sequence demonstrates how to enable the instruction and data cache for the current Security state when running in privileged mode.

```
CCR EQU 0xE000ED14
LDR R0, =CCR ; Read CCR
LDR r1, [R0] ; Set bits 16 and 17 to enable D-cache and I-cache
ORR R1, R1, #(0x3 << 16)
STR R1, [R0] ; Write back the modified value to the CCR
DSB
ISB ; Perform DSB and ISB to guarantee change is visible to subsequent instructions
```

4.5.14 Powering down the caches

To powerdown the caches:

1. Set CCR.DC and CCR.IC to 0. CPDLPSTATE.RLPSTATE must be set to `0b11`.
2. If the data cache contains dirty data that must be transferred to system memory, the entire cache must be cleaned with a set of Set/Way cache maintenance operations.

```
CCSIDR EQU 0xE000ED80 ; Cache size ID register address
CSSELR EQU 0xE000ED84 ; Cache size selection register address
DCCSW  EQU 0xE000EF6C ; Cache maintenance op address: data cache clean by set/way
; CSSELR selects the cache visible in CCSIDR
MOV r0, #0x0 ; 0 = select “level 1 data cache”
LDR r11, =CSSELR ;
STR r0, [r11] ;
DSB ; Ensure write to CSSELR before proceeding
LDR r11, =CCSIDR ; From CCSIDR
LDR r2, [r11] ; Read data cache size information
AND r1, r2, #0x7 ; r1 = cache line size
ADD r7, r1, #0x4 ; r7 = number of words in a cache line
UBFX r4, r2, #3, #10 ; r4 = number of “ways”-1 of data cache
UBFX r2, r2, #13, #15 ; r2 = number of “set”-1 of data cache
CLZ r6, r4 ; calculate bit offset for “way” in DCISW
LDR r11, =DCCSW ; clean cache by set/way
inv_loop1 ; For each “set”
MOV r1, r4 ; r1 = number of “ways”-1
LSLS r8, r2, r7 ; shift “set” value to bit 5 of r8
inv_loop2 ; For each “way”
LSLS r3, r1, r6 ; shift “way” value to bit 30 in r6
ORRS r3, r3, r8 ; merge “way” and “set” value for DCISW
STR r3, [r11] ; invalidate D-cache line
SUBS r1, r1, #0x1 ; decrement “way”
BGE inv_loop2 ; End for each “way”
SUBS r2, r2, #0x1 ; Decrement “set”
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

```
BGE inv_loop1 ; End for each "set"
DSB ; Data sync barrier after invalidate cache
ISB ; Instruction sync barrier after invalidate cache
```

3. Set MSCR.DCACTIVE and MSCR.ICACTIVE to 0. As a result, the processor core deasserts bit 16 of the **COREPACTIVE** signal, which is a hint to the external power controller that PDRAMS can be powered down.

4.5.15 Powering up the caches

To powerup the caches:

1. Set MSCR.DCACTIVE and MSCR.ICACTIVE to 1. As a result, the processor core asserts **COREPACTIVE[16]**, to indicate to an external power controller that PDRAMS can be powered up.
2. Set CCR.DC and CCR.IC to 1. After the external power control logic has powered up PDRAMS, the *Core Power Control* (CPC) triggers an automatic invalidation of the RAMs (if **INITL1RSTDIS** is 0), and after that is complete, subsequent instructions can cause allocations to and lookups in the caches.

4.5.16 Enabling the branch cache

The branch cache is disabled on reset. You must enable the branch cache to implement *Low Overhead Branch* (LOB) Extension.

The processor core must be in privileged mode to read from and write to the CCR. If the Security Extension is implemented, the CCR.LOB bit is banked so it must be enabled for each Security state that uses the LOB Extension. For more information on CCR, see the *Arm®v8-M Architecture Reference Manual*.

The following code sequence demonstrates how to enable the branch cache for the current Security state when running in privileged mode.

```
CCR EQU 0xE000ED14
LDR R0, =CCR ; Read CCR
LDR r1, [R0] ; Set bits 19 to enable LOB
ORR R1, R1, #(0x8 << 16)
STR R1, [R0] ; Write back the modified value to the CCR
DSB
ISB ; Reset pipeline now LOB is enabled.
```

4.5.17 Fault handling considerations

Cache maintenance operations can result in a BusFault. Such fault events are asynchronous.

This type of BusFault:

- Does not cause escalation to HardFault where a BusFault handler is enabled.
- Never causes lockup.

Because the fault event is asynchronous, software code for cache maintenance operations must use memory barrier instructions, such as `DSB`, on completion so that the fault event can be observed immediately.

4.5.18 Cache maintenance design

You must always place a DSB and ISB instruction sequence after a cache maintenance operation to ensure that the effect is observed by any following instructions in the software.

When using a cache maintenance operation by address or set/way a `DSB` instruction must be executed after any previous load or store, and before the maintenance operation, to guarantee that the effect of the load or store is observed by the operation. For example, if a store writes to the address accessed by a DCCMVAC the DSB instruction guarantees that the dirty data is correctly cleaned from the data cache.

When one or more maintenance operations have been executed, use of a `DSB` instruction guarantees that they have completed and that any following load or store operations executes in order after the maintenance operations.

Cache maintenance operations always complete in-order with respect to each other. This means only one `DSB` instruction is required to guarantee the completion of a set of maintenance operations.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following code sequence shows how to use cache maintenance operations to synchronize the data and instruction caches for self-modifying code. The sequence is entered with <Rx> containing the new 32-bit instruction. Use `STRH` in the first line instead of `STR` for a 16-bit instruction:

```
STR <Rx>, <inst_address1>
DSB ; Ensure the data has been written to the cache.
STR <inst_address1>, DCCMVAU ; Clean data cache by MVA to point of unification (PoU).
STR <inst_address1>, ICIMVAU ; Invalidate instruction cache by MVA to PoU.
DSB ; Ensure completion of the invalidations.
ISB ; Synchronize fetched instruction stream.
```


Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.6 Memory Authentication

The processor can use security attribution and memory protection to manage sensitive data. The *Security Attribution Unit* (SAU) and *Memory Protection Unit* (MPU) are found in the *Memory Authentication Unit* (MAU), which receives requests from units that perform memory accesses and returns responses accordingly.

These responses are a combination of all responses from the following units:

- *Security Attribution Unit* (SAU).
- *Implementation Defined Attribution Unit* (IDAU).
- *Memory Protection Unit* (MPU).
- *TCM Gate Unit* (TGU).

4.6.1 Security Attribution Unit

The processor uses a *Security Attribution Unit* (SAU) to determine the security of an address.

SAU features

- The number of regions that are included in the SAU are configured in the Cortex-M55 implementation to be 0, 4, or 8.
- The SAU is not used for *Slave AHB* (S-AHB) accesses.

SAU features

If the Security Extension is included, you can optionally include an SAU for security attribution checking.

- The number of regions that are included in the SAU are configured in the Cortex-M55 implementation to be 0, 4, or 8.
- The SAU is not used for *Slave AHB* (S-AHB) accesses.

——— **Note** ———

The processor has an external signal, **LOCKSAU**, that

disables writes to registers that are associated with the *Security Attribution Unit* (SAU) region from software or from a debug agent connected to the processor.

- SAU_CTRL.
- SAU_RNR.
- SAU_RBAR.
- SAU_RLAR.

———————————

Security Attribution Unit register summary

The *Security Attribution Unit* (SAU) has various registers associated to its function. Each of these registers is 32 bits wide.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table shows a summary of the SAU registers.

Table 4-78 SAU registers summary

Address	Name	Type	Reset value	Description
`0xE000EDD0`	SAU_CTRL	RW	`0x00000000`	See *Security Attribution Unit Control Register* on page 4-556. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI.
`0xE000EDD4`	SAU_TYPE	RO	`0x0000000X` **Note** SAU_TYPE[3:0] depends on the number of SAU regions included. This value can be 0,4, or 8.	See *Security Attribution Unit Type Register* on page 4-557. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI. <Add reference to the number of SAU regions in your implementation reflected in SAU_TYPE [3:0]>
`0xE000EDD8`	SAU_RNR	RW	UNKNOWN	See *Security Attribution Unit Region Number Register* on page 4-558. In Non-secure state, this register is RAZ/WI.
`0xE000EDDC`	SAU_RBAR	RW	UNKNOWN	See *Security Attribution Unit Region Base Address Register* on page 4-559. In Non-secure state, this register is RAZ/WI.
`0xE000EDE0`	SAU_RLAR	RW	Bit[0] resets to 0. Other bits reset to an UNKNOWN value.	See *Security Attribution Unit Region Limit Address Register* on page 4-559. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI.
`0xE000EDE4`	SFSR	RW	`0x00000000`	See *Secure Fault Status Register* on page 4-560. In Non-secure state, this register is RAZ/WI.
`0xE000EDE8`	SFAR	RW	UNKNOWN	See *Secure Fault Address Register* on page 4-562. In Non-secure state, this register is RAZ/WI.

Note

- Only Privileged accesses to the SAU registers are permitted. Unprivileged accesses generate a fault.
- The SAU registers are word accessible only. Halfword and byte accesses are UNPREDICTABLE.
- The SAU registers are RAZ/WI when accessed from Non-secure state.
- The SAU registers are not banked between Security states.

Security Attribution Unit Control Register

The SAU_CTRL allows enabling of the *Security Attribution Unit* (SAU)

Usage constraints

See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration

This register is always implemented.

Attributes

This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following figure shows the SAU_CTRL bit assignments.

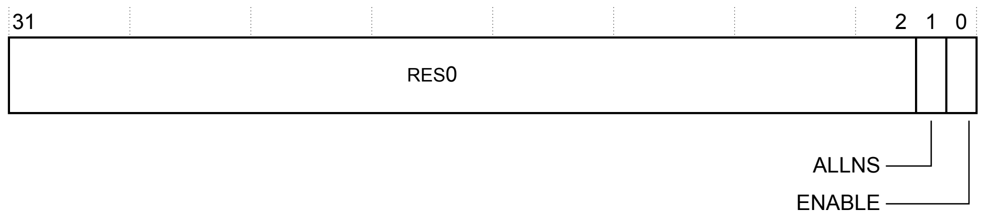


Figure 4-48 SAU_CTRL bit assignments

The following table describes the SAU_CTRL bit assignments.

Table 4-79 SAU_CTRL bit assignments

Bits	Name	Function
[31:2]	-	Reserved, RES0.
[1]	ALLNS	All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls whether 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 has no effect when SAU_ENABLE is 1. If the number of SAU regions is or if the SAU is not enabled, then to use the IDAU, this bit must be set to 1. Setting SAU_CTRL.ALLNS to 1 allows the security attribution of all addresses to be set by the IDAU in the system.
[0]	ENABLE	Enables the (SAU). The possible values of this bit are: **0** Unit is disabled. **1** Unit is enabled. This bit is RAZ/WI when the Security Extension is implemented without an (SAU) region.

Security Attribution Unit Type Register

The SAU_TYPE indicates the number of regions implemented by the *Security Attribution Unit* (SAU).

Usage constraints
See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration
This register is always implemented.

Attributes
This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.
In an implementation with the Security Extension, this register is:
- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SAU_TYPE bit assignments.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

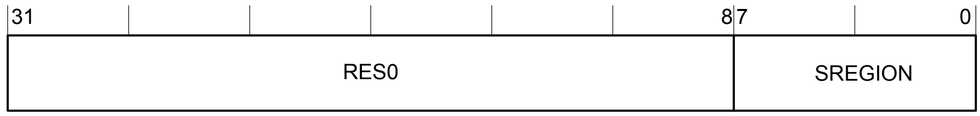


Figure 4-49 SAU_TYPE bit assignments

The following table describes the SAU_TYPE bit assignments.

Table 4-80 SAU_TYPE bit assignments

Bits	Name	Function
[31:8]	-	Reserved, RES0.
[7:0]	SREGION	The number of implemented (SAU) regions. <Specify implementation-specific SAU regions.>

Security Attribution Unit Region Number Register

The SAU_RNR selects the region currently accessed by SAU_RBAR and SAU_RLAR.

Usage constraints

See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration

This register is always implemented.

Attributes

This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.
In an implementation with the Security Extension, this register is:
- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SAU_RNR bit assignments.

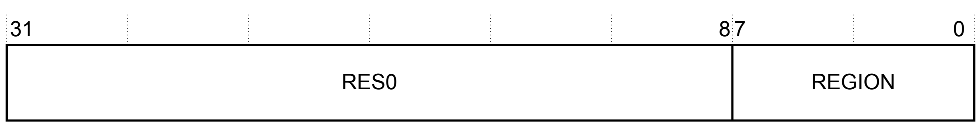


Figure 4-50 SAU_RNR bit assignments

Table 4-81 SAU_RNR bit assignments

Bits	Name	Function
[31:8]	-	Reserved, RES0.
[7:0]	REGION	Region number. Indicates the *Security Attribution Unit* (SAU) region accessed by SAU_RBAR and SAU_RLAR. If no regions are implemented, this field is reserved. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE. This field resets to an UNKNOWN value on a Warm reset. <Specify implementation-specific region number settings.>

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Security Attribution Unit Region Base Address Register

The SAU_RBAR provides indirect read and write access to the base address of the currently selected *Security Attribution Unit* (SAU) region.

Usage constraints

See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration

This register is always implemented.

Attributes

This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SAU_RBAR bit assignments.

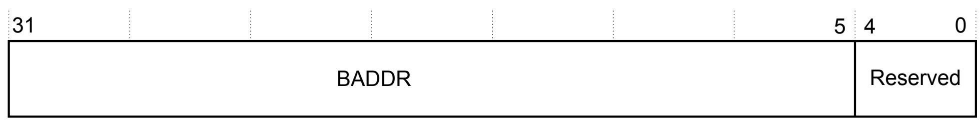


Figure 4-51 SAU_RBAR bit assignments

The following table describes the SAU_RBAR bit assignments.

Table 4-82 SAU_RBAR bit assignments

Bits	Name	Function
[31:5]	BADDR	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`.
[4:0]	-	Reserved, RES0.

Security Attribution Unit Region Limit Address Register

The SAU_RLAR provides indirect read and write access to the limit address of the currently selected *Security Attribution Unit* (SAU) region.

Usage constraints

See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration

This register is always implemented.

Attributes

This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SAU_RLAR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

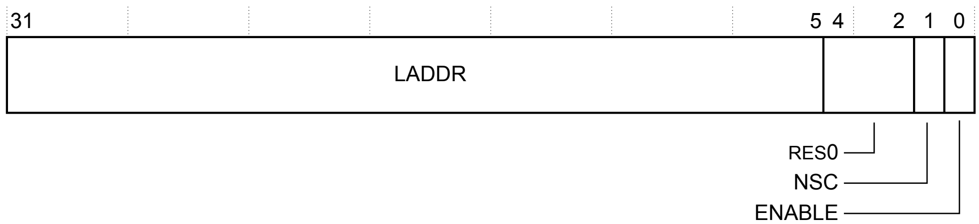


Figure 4-52 SAU_RLAR bit assignments

The following table describes the SAU_RLAR bit assignments.

Table 4-83 SAU_RLAR bit assignments

Bits	Name	Function
[31:5]	LADDR	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`.
[4:2]	-	Reserved, RES0.
[1]	NSC	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.
[0]	ENABLE	Enable. (SAU) region enable. The possible values of this bit are: **0** Region is disabled. **1** Region is enabled. This bit reset to 0 on a Warm reset.

Secure Fault Status Register

The SFSR provides information about any security related faults.

Usage constraints
See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration
This register is always implemented.

Attributes
This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.
In an implementation with the Security Extension, this register is:
- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SFSR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

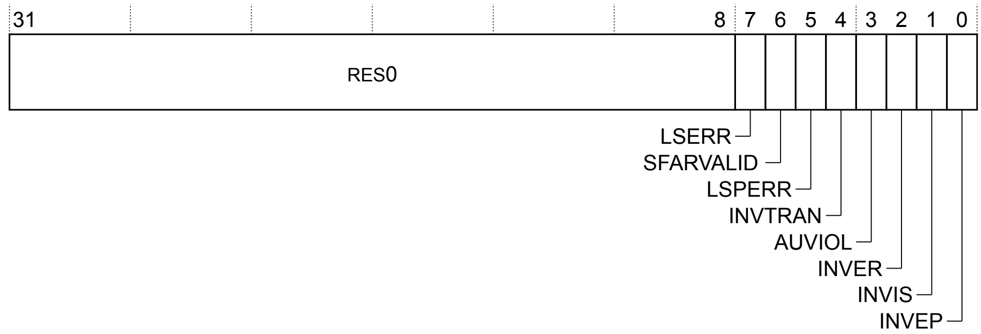


Figure 4-53 SFSR bit assignments

The following table describes the SFSR bit assignments.

Table 4-84 SFSR bit assignments

Bits	Name	Function
[31:8]	-	Reserved, RES0.
[7]	LSERR	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.
[6]	SFARVALID	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.
[5]	LSPERR	Lazy state preservation error flag. Stick flag indicating that an *Security Attribution Unit* (SAU) or *Implementation Defined Attribution Unit* (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.
[4]	INVTRAN	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.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-84 SFSR bit assignments (continued)

Bits	Name	Function
[3]	AUVIOL	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.
[2]	INVER	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.
[1]	INVIS	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.
[0]	INVEP	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.

Secure Fault Address Register

The SFSR shows the address of the memory location that caused a security violation.

Usage constraints

See *Security Attribution Unit register summary* on page 4-555 for the SAU_CTRL attributes.

Configuration

This register is always implemented.

Attributes

This register is RAZ/WI when accessed as Non-secure. This register is not banked between Security states.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The following figure shows the SFAR bit assignments.

Figure 4-54 SFAR bit assignments

The following table shows the SFAR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-85 SFAR bit assignments

Bits	Name	Function
[31:0]	ADDRESS	When the SFARVALID bit of the SFSR is set to 1, this field holds the address of an access that caused a *Security Attribution Unit* (SAU) violation.

4.6.2 Memory Protection Unit

The MPU is divided into four, eight, 12, or 16 regions and defines the location, size, access permissions, and memory attributes of each region.

The MPU supports:

- Independent attribute settings for each region.
- Export of memory attributes to the system.

If the processor implements the Security Extension, it contains:

- One optional Secure MPU.
- One optional Non-secure MPU.

When memory regions overlap, the processor generates a fault if a core access hits the overlapping regions.

The MPU memory map is unified. This means instruction accesses and data accesses have the same region settings.

If a program accesses a memory location that is prohibited by the MPU, the processor generates a MemManage exception.

In an OS environment, the kernel can update the MPU region setting dynamically based on the process to be executed. Typically, an embedded OS uses the MPU for memory protection.

Configuration of MPU regions is based on memory types, see *2.2.2 Memory regions, types, and attributes* on page 2-51.

The following table shows the possible MPU region attributes. If the processor is configured without a cache, these include Shareability and cache behavior attributes that are not relevant to most microcontroller implementations.

See *MPU configuration for a microcontroller* on page 4-573 for guidelines for programming such an implementation.

Table 4-86 Memory attributes summary

Memory type	Shareability	Other attributes	Description
Device-nGnRnE	Shareable	-	Used to access memory mapped peripherals.All accesses to Device-nGnRnE memory occur in program order. All regions are assumed to be shared.
Device-nGnRE	Shareable	-	Used to access memory mapped peripherals.Weaker ordering than Device-nGnRnE.
Device-nGRE	Shareable	-	Used to access memory mapped peripherals.Weaker ordering than Device-nGnRE.
Device-GRE	Shareable	-	Used to access memory mapped peripherals.Weaker ordering than Device-nGRE.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-86 Memory attributes summary (continued)

Memory type	Shareability	Other attributes	Description
Normal	Shareable	Non-cacheable Write-Through Cacheable Write-Back Cacheable	Normal memory that is shared between several processors.
Normal	Non-Shareable	Non-cacheable Write-Through Cacheable Write-Back Cacheable	Normal memory that only a single processor uses.

Note

The processor has an external signal, **LOCKSMPU**, that disables writes to registers that are associated with the Secure *Memory Protection Unit* (MPU) region from software or from a debug agent connected to the processor.

- MPU_CTRL.
- MPU_RNR.
- MPU_RBAR.
- MPU_RLAR.
- MPU_RBAR_An.
- MPU_RLAR_An.

The processor has an external signal, **LOCKNSMPU**, that disables writes to registers that are associated with the Non-secure MPU region from software or from a debug agent connected to the processor.

- MPU_CTRL_NS.
- MPU_RNR_NS.
- MPU_RBAR_NS.
- MPU_RLAR_NS.
- MPU_RBAR_A_NSn.
- MPU_RLAR_A_NSn.

Memory Protection Unit register summary

Use the 32-bit *Memory Protection Unit* (MPU) registers to define the MPU regions and their attributes.

The following table shows a summary of the MPU registers.

Table 4-87 MPU registers summary

Address	Name	Non-secure alias address	Non-secure alias name	Type	Reset Value	Description
`0xE000ED90`	MPU_TYPE	`0xE002ED90`	MPU_TYPE_NS	RO	The reset value is fixed and depends on the value of bits[15:8] and implementation options. This value can be 0, 4, 8, 12, or 16.	See *MPU Type Register* on page 4-565.
`0xE000ED94`	MPU_CTRL	`0xE002ED94`	MPU_CTRL_NS	RW	`0x00000000`	See *MPU Control Register* on page 4-566.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-87 MPU registers summary (continued)

Address	Name	Non-secure alias address	Non-secure alias name	Type	Reset Value	Description
0xE000ED98	MPU_RNR	0xE002ED98	MPU_RNR_NS	RW	0x000000XX	See *MPU Region Number Register* on page 4-567.
0xE000ED9C	MPU_RBAR	0xE002ED9C	MPU_RBAR_NS	RW	UNKNOWN	See *MPU Region Base Address Register* on page 4-568.
0xE000EDA0	MPU_RLAR	0xE002EDA0	MPU_RLAR_NS	RW	UNKNOWN	See *MPU Region Limit Address Register* on page 4-569.
0xE000EDA4	MPU_RBAR_A<n>	0xE002EDA4	MPU_RBAR_A<n>_NS	RW	UNKNOWN	See *MPU Region Base Address Register Alias, n=1-3* on page 4-569
0xE000EDA8	MPU_RLAR_A<n>	0xE002EDA8	MPU_RLAR_A<n>_NS	RW	UNKNOWN	See *MPU Region Limit Address Register Alias, n=1-3* on page 4-570.
0xE000EDC0	MPU_MAIR0	0xE002EDC0	MPU_MAIR0_NS	RW	UNKNOWN	See *MPU Memory Attribute Indirection Registers 0 and 1* on page 4-570.
0xE000EDC4	MPU_MAIR1	0xE002EDC4	MPU_MAIR1_NS	RW	UNKNOWN	

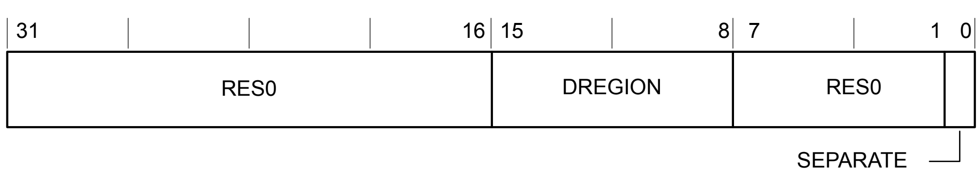
MPU Type Register

The MPU_TYPE register indicates whether the MPU is present, and if so, how many regions it supports.

This register is banked between Security states.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_TYPE bit assignments are:



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-88 MPU_TYPE bit assignments

Bits	Name	Function
[31:16]	-	Reserved, RES0.
[15:8]	DREGION	Data regions. Number of regions supported by the MPU. `0x00` Zero regions if your device does not include the MPU. `0x04` Four regions if your device includes the MPU. This value is implementation defined. `0x08` Eight regions if your device includes the MPU. This value is implementation defined. `0x0C` 12 regions if your device includes the MPU. This value is implementation defined. `0x10` 16 regions if your device includes the MPU. This value is implementation defined.
[7:1]	-	Reserved, RES0.
[0]	SEPARATE	Indicates support for unified or separate instructions and data address regions. Armv8-M only supports unified MPU regions. **0** Unified.

MPU Control Register

The MPU_CTRL register enables the MPU.

When the MPU is enabled, it controls whether the default memory map is enabled as a background region for privileged accesses and whether the MPU is enabled for HardFaults, and NMIs.

This register is banked between Security states.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_CTRL bit assignments are:

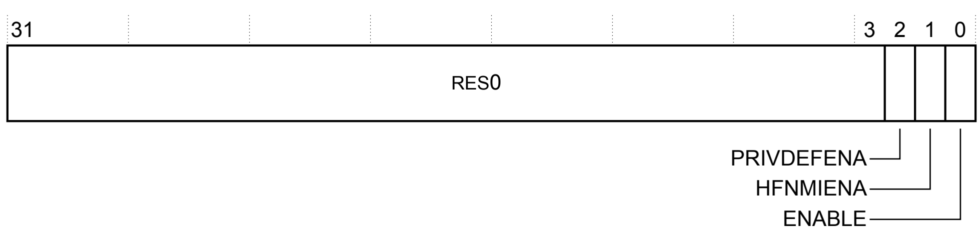


Table 4-89 MPU_CTRL bit assignments

Bits	Name	Function
[31:3]	-	Reserved, RES0.
[2]	PRIVDEFENA	Enables privileged software access to the default memory map. When the MPU is enabled: **0** Disables use of the default memory map. Any memory access to a location that is not covered by any enabled region causes a fault. **1** Enables use of the default memory map as a background region for privileged software accesses. When enabled, the background region acts as if it has the lowest priority. Any region that is defined and enabled has priority over this default map. If the MPU is disabled, the processor ignores this bit.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-89 MPU_CTRL bit assignments (continued)

Bits	Name	Function
[1]	HFNMIENA	Enables the operation of MPU during HardFault and NMI handlers. When the MPU is enabled: **0** MPU is disabled during HardFault and NMI handlers, regardless of the value of the ENABLE bit. **1** The MPU is enabled during HardFault and NMI handlers. When the MPU is disabled, if this bit is set to 1 the behavior is UNPREDICTABLE.
[0]	ENABLE	Enables the MPU: **0** MPU is disabled. **1** MPU is enabled.

XN and Device-nGnRnE rules always apply to the System Control Space regardless of the value of the ENABLE bit.

When the ENABLE bit is set to 1, at least one region of the memory map must be enabled for the system to function unless the PRIVDEFENA bit is set to 1. If the PRIVDEFENA bit is set to 1 and no regions are enabled, then only privileged software can operate.

When the ENABLE bit is set to 0, the system uses the default memory map. This has the same behavior as if the MPU is not implemented.

The default memory map applies to accesses from both privileged and unprivileged software.

When the MPU is enabled, accesses to the System Control Space and vector table are always permitted. Other areas are accessible based on regions and whether PRIVDEFENA is set to 1.

Unless HFNMIENA is set to 1, the MPU is not enabled when the processor is executing the handler for an exception with priority –1, –2, or –3. These priorities are only possible when handling a HardFault or NMI exception. Setting the HFNMIENA bit to 1 enables the MPU when operating with these priorities.

MPU Region Number Register

The MPU_RNR selects the region currently accessed by MPU_RBAR and MPU_RLAR.

Usage constraints

See *Memory Protection Unit register summary* on page 4-564 for the MPU_RNR attributes.

Configurations

This register is always implemented.

Attributes

This register is banked between Security states.
In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the MPU_RNR bit assignments.

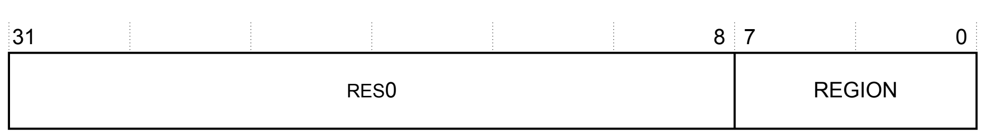


Figure 4-55 MPU_RNR bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table describes the MPU_RNR bit assignments.

Table 4-90 MPU_RNR bit assignments

Bits	Name	Function
[31:8]	-	Reserved, RES0.
[7:0]	REGION	Regions. Indicates the memory region accessed by MPU_RBAR and PMU_RLAR. If no MPU region is implemented, this field is reserved. Writing a value corresponding to an unimplemented region is CONSTRAINED UNPREDICTABLE.

You must write the required region number to this register before accessing the MPU_RBAR or MPU_RLAR.

MPU Region Base Address Register

The MPU_RBAR defines the base address of the MPU region selected by the MPU_RNR.

Usage constraints
See *Memory Protection Unit register summary* on page 4-564 for the MPU_RBAR attributes.

Configurations
This register is always implemented.

Attributes
This register is banked between Security states.
In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the MPU_RBAR bit assignments.

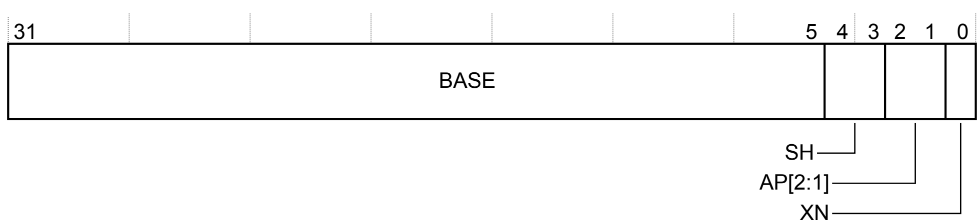


Figure 4-56 MPU_RBAR bit assignments

The following table describes the MPU_RBAR bit assignments.

Table 4-91 MPU_RBAR bit assignments

Bits	Name	Function
[31:5]	BASE	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.
[4:3]	SH	Shareability. Defines the shareability domain of this region for Normal memory. `0b00` Non-shareable. `0b01` UNPREDICTABLE. `0b10` Outer shareable. `0b11` Inner Shareable. All other values are reserved. For any type of Device memory, the value of this field is ignored.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-91 MPU_RBAR bit assignments (continued)

Bits	Name	Function
[2:1]	AP[2:1]	Access permissions. `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.
[0]	XN	Execute Never. Defines whether code can be executed from this region. **0** Execution only permitted if read permitted. **1** Execution not permitted.

MPU Region Base Address Register Alias, n=1-3

The MPU_RBAR_A<n> provides indirect read and write access to the MPU base address register. Accessing MPU_RBAR_A<n> is equivalent to setting MPU_RNR[7:2]:n[1:0] and then accessing MPU_RBAR for the Security state.

MPU Region Limit Address Register

The MPU_RLAR defines the limit address of the MPU region selected by the MPU_RNR.

Usage constraints
See *Memory Protection Unit register summary* on page 4-564 for the MPU_RLAR attributes.

Configurations
This register is always implemented.

Attributes
This register is banked between Security states.
In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the MPU_RLAR bit assignments.

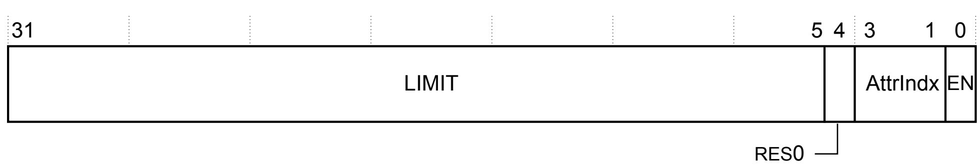


Figure 4-57 MPU_RLAR bit assignments

The following table describes the MPU_RLAR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-92 MPU_RLAR bit assignments

Bits	Name	Function
[31:5]	LIMIT	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.
[4]	PXN	Privileged execute never. Defines whether code can be executed from this privileged region. **0** Execution only permitted if read permitted. **1** Execution from a privileged mode is not permitted.
[3:1]	AttrIndx	Attribute index. Associates a set of attributes in the MPU_MAIR0 and MPU_MAIR1 fields.
[0]	EN	Enable. Region enable. The possible values of this bit are: **0** Region disabled. **1** Region enabled.

MPU Region Limit Address Register Alias, n=1-3

The MPU_RLAR_A<n> provides indirect read and write access to the MPU limit address register. Accessing MPU_RLAR_A<n> is equivalent to setting MPU_RNR[7:2]:n[1:0] and then accessing MPU_RLAR for the Security state

MPU Memory Attribute Indirection Registers 0 and 1

The MPU_MAIR0 and MPU_MAIR1 provide the memory attribute encodings corresponding to the AttrIndex values.

Usage constraints
See *Memory Protection Unit register summary* on page 4-564 for the MPU_RLAR attributes.

Configurations
This register is always implemented.

Attributes
This register is banked between Security states.
In an implementation with the Security Extension, this register is banked between Security states.

The following figure shows the MPU_MAIR0 bit assignments.

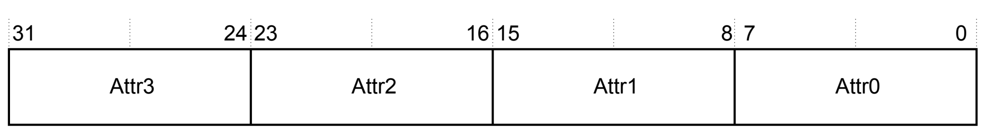
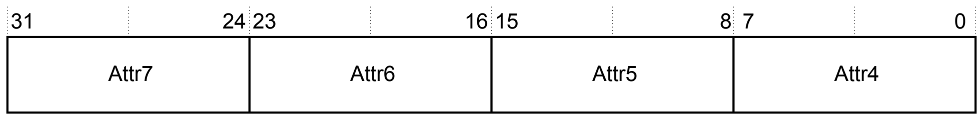


Figure 4-58 MPU_MAIR0 bit assignments

Attr<n>, bits [8n+7:8n], for n= 0 to 3.
Memory attribute encoding for MPU regions with an AttrIndex of n.

The MPU_MAIR1 bit assignments are:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



Attr<n>, bits [8(n-4)+7:8(n-4)], for n = 4 to 7

Memory attribute encoding for MPU regions with an AttrIndex of n.

MAIR_ATTR defines the memory attribute encoding used in MPU_MAIR0 and MPU_MAIR1, and the bit assignments are:

When MAIR_ATTR[7:4] is `0000`:

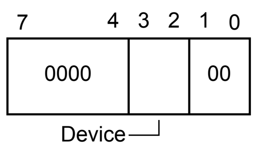
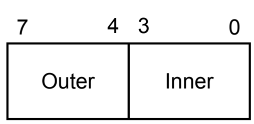


Table 4-93 MAIR_ATTR values for bits[3:2] when MAIR_ATTR[7:4] is 0000

Bits	Name	Function
[3:2]	Device	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.

When MAIR_ATTR[7:4] is not `0000`:



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-94 MAIR_ATTR bit assignments when MAIR_ATTR[7:4] is not 0000

Bits	Name	Function
[7:4]	Outer	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 is not `00`). `0b0100` Normal memory, Outer non-cacheable. `0b01RW` Normal memory, Outer write-back transient (RW is not `00`). `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.
[3:0]	Inner	Inner attributes. Specifies the Inner memory attributes. The possible values of this field are: `0b0000` UNPREDICTABLE. `0b00RW` Normal memory, Inner Write-Through Transient (RW is not `00`). `0b0100` Normal memory, Inner non-cacheable. `0b01RW` Normal memory, Inner write-back transient (RW is not `00`). `0b10RW` Normal memory, Inner write-through non-transient. `0b11RW` Normal memory, Inner write-back non-transient. R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

Updating protected memory regions

To update an MPU region, update the attributes in the MPU_RNR, MPU_RBAR and MPU_RLAR registers. To update an SAU region, update the attributes in the SAU_RNR, SAU_RBAR and SAU_RLAR registers.

Updating an MPU region

Simple code to configure one region:

```
; R1 = MPU region number
; R2 = base address, permissions and shareability
; R3 = limit address, attributes index and enable
LDR R0,=MPU_RNR
STR R1, [R0, #0x0]   ; MPU_RNR
STR R2, [R0, #0x4]   ; MPU_RBAR
STR R3, [R0, #0x8]   ; MPU_RLAR
```

Software must use memory barrier instructions:

- Before MPU setup if there might be outstanding memory transfers, such as buffered writes, that might be affected by the change in MPU settings.
- After MPU setup if it includes memory transfers that must use the new MPU settings.

If you want all the MPU memory access behavior to take effect immediately after the programming sequence, use a `DSB` instruction and an `ISB` instruction.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Updating an SAU region

Simple code to configure one region:

```
; R1 = SAU region number
; R2 = base address
; R3 = limit address, Non-secure callable attribute and enable
LDR R0,=SAU_RNR
STR R1, [R0, #0x0]   ; SAU_RNR
STR R2, [R0, #0x4]   ; SAU_RBAR
STR R3, [R0, #0x8]   ; SAU_RLAR
```

Software must use memory barrier instructions:

- Before SAU setup if there might be outstanding memory transfers, such as buffered writes, that might be affected by the change in SAU settings.
- After SAU setup if it includes memory transfers that must use the new SAU settings.

If you want all the SAU memory access behavior to take effect immediately after the programming sequence, use a `DSB` instruction and an `ISB` instruction.

MPU design hints and tips

To update the attributes for an MPU region, update the MPU_RNR, MPU_RBAR, and MPU_RLAR registers.

To avoid unexpected behavior, disable the interrupts before updating the attributes of a region that the interrupt handlers might access. When setting up the MPU, and if the MPU has previously been programmed, disable unused regions to prevent any previous region settings from affecting the new MPU setup.

MPU configuration for a microcontroller

Usually, a microcontroller system has only a single processor and the instruction and data caches.

In such a system, program the MPU as follows:

Table 4-95 Memory region attributes for a microcontroller

Memory region	MAIR_ATTR.Outer	MAIR_ATTR.Inner	Shareability	Memory type and attributes	Comments
Flash memory	!=`0b0000`	`0b1010`	0	Normal memory, Non-shareable, Write-Through.	Typically used by the instruction cache.
Internal SRAM	!=`0b0000`	`0b1010`	1	Normal memory, Shareable, Write-Through.	Typically used by the data cache configured with ECC.
External SRAM	!=`0b0000`	`0b1111`	1	Normal memory, Shareable, Write-Back	Typically used by the data cache configured without ECC.
Peripherals	`0b0000`	-	-	Device memory	-

However, using these settings for the MPU regions makes the application code more portable. The values given are for typical situations. In special systems, such as multiprocessor designs or designs with a separate DMA engine, the shareability attribute might be important. In these cases, refer to the recommendations of the memory device manufacturer.

Shareability attributes define whether the global monitor is used, or only the local monitor is used.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.6.3 Implementation Defined Attribution Unit

The processor supports an external *Implementation Defined Attribution Unit* (IDAU) to allow the system determine the security level associated with any given address.

IDAU features

- The processor has three external interfaces for the IDAU with identical signals, properties, and requirements.
  — An interface for instruction fetches and exception vector read operations.
  — Two interfaces for all other data read and write operations from load and store instructions, register stacking on exception entry and exit, and debug memory accesses.
- The IDAU is not used for *Slave AHB* (S-AHB) accesses.

IDAU features

- The processor has three external interfaces for the IDAU with identical signals, properties, and requirements.
  — An interface for instruction fetches and exception vector read operations.
  — Two interfaces for all other data read and write operations from load and store instructions, register stacking on exception entry and exit, and debug memory accesses.
- It is only implemented if the Armv8.1-M Security Extension is included in the processor.
- The IDAU is not used for *Slave AHB* (S-AHB) accesses.

Security levels

The security level that the *Memory Authentication Unit* (MAU) returns is a combination of the region type defined in the internal SAU, if configured, and the security type from the IDAU. For more information, see *4.6.1 Security Attribution Unit* on page 4-555.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.7 Implementation defined register summary

The 32-bit IMPLEMENTATION DEFINED registers are either IMPLEMENTATION DEFINED in the architecture or located in the IMPLEMENTATION DEFINED memory space in the architecture. These registers provide memory configuration and access control, error record information, interrupt control, and processor configuration information.

The following table lists the IMPLEMENTATION DEFINED registers.

Table 4-96 IMPLEMENTATION DEFINED register summary

Address	Name	Type	Reset value	Description
0xE0005000	ERRFR0	RO	0x00000101	*5.6.1 Error Record Feature Register, ERRFR0* on page 5-643
0xE0005008	ERRCTRL0	-	-	This register is RES0.
0xE0005010	ERRSTATUS0	RW	UNKNOWN	*5.6.2 Error Record Primary Status Register, ERRSTATUS0* on page 5-644
0xE0005018	ERRADDR0	RO	UNKNOWN	*5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20* on page 5-646
0xE000501C	ERRADDR20	RO	UNKNOWN	*5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20* on page 5-646
0xE0005020	ERRMISC00	-	-	This register is RES0.
0xE0005024	ERRMISC10	RO	UNKNOWN	*5.6.4 Error Record Miscellaneous Register 10, ERRMISC10* on page 5-648
0xE0005028	ERRMISC20	-	-	This register is RES0.
0xE000502C	ERRMISC30	-	-	This register is RES0.
0xE0005030	ERRMISC40	-	-	This register is RES0.
0xE0005034	ERRMISC50	-	-	This register is RES0.
0xE0005038	ERRMISC60	-	-	This register is RES0.
0xE000503C	ERRMISC70	-	-	This register is RES0.
0xE0005E00	ERRGSR0	RO	0x00000000	*5.6.5 Fault Group Status Register, ERRGSR0* on page 5-649

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-96 IMPLEMENTATION DEFINED register summary (continued)

Address	Name	Type	Reset value	Description
0xE0005FC8	ERRDEVID	RO	0x00000001	*5.6.6 Error Record Device ID Register, ERRDEVID* on page 5-650
0xE000E008	ACTLR	RW	0x00000000	*4.3.38 Auxiliary Control Register* on page 4-530
0xE000ED3C	AFSR	RW	0x00000000	*4.3.2 Auxiliary Fault Status Register* on page 4-476
0xE000EF04	RFSR	RW	UNKNOWN	*5.6.7 Fault Status Register, RFSR* on page 5-650
0xE001E000	MSCR	RW	UNKNOWN	*4.8.2 Memory System Control Register, MSCR* on page 4-585
0xE001E004	PFCR	RW	0x00000061	*4.8.4 Prefetcher Control Register, PFCR* on page 4-587
0xE001E010	ITCMCR	RW	UNKNOWN	*4.8.5 TCM Control Registers, ITCMCR and DTCMCR* on page 4-588
0xE001E014	DTCMCR	RW	UNKNOWN	
0xE001E018	PAHBCR	RW	UNKNOWN	*4.8.3 P-AHB Control Register, PAHBCR* on page 4-586
0xE001E100	IEBR0	RW	0x00000000	*4.10.1 Instruction Cache Error Bank Register 0-1, IEBR0 and IEBR1* on page 4-598
0xE001E104	IEBR1	RW	0x00000000	
0xE001E110	DEBR0	RW	0x00000000	*4.10.2 Data Cache Error Bank Register 0-1, DEBR0 and DEBR1* on page 4-599
0xE001E114	DEBR1	RW	0x00000000	
0xE001E120	TEBR0	RW	0x00000000	*4.10.3 TCM Error Bank Register 0-1, TEBR0 and TEBR1* on page 4-601

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-96 IMPLEMENTATION DEFINED register summary (continued)

Address	Name	Type	Reset value	Description
0xE001E124	TEBRDATA0	RO	0x00000000	*Data for TCU Error Bank Register 0-1, TEBRDATA0 and TEBRDATA1* on page 4-602
0xE001E128	TEBR1	RW	0x00000000	*4.10.3 TCM Error Bank Register 0-1, TEBR0 and TEBR1* on page 4-601
0xE001E12C	TEBRDATA1	RO	0x00000000	*Data for TCU Error Bank Register 0-1, TEBRDATA0 and TEBRDATA1* on page 4-602
0xE001E200	DCADCRR	RO	UNKNOWN	*Direct Cache Access Read Registers, DCAICRR and DCADCRR* on page 4-582
0xE001E204	DCAICRR	RO	UNKNOWN	
0xE001E210	DCADCLR	RW	0x00000000	*Direct Cache Access Location Registers, DCAICLR and DCADCLR* on page 4-580
0xE001E214	DCAICLR	RW	0x00000000	
0xE001E300	CPDLPSTATE	RW	0x00000333	*4.9.1 Core Power Domain Low Power State Register, CPDLPSTATE* on page 4-595
0xE001E304	DPDLPSTATE	RW	0x00000003	*4.9.2 Debug Power Domain Low Power State Register, DPDLPSTATE* on page 4-596
0xE001E400	EVENTSPR	WO	UNKNOWN	*4.13.1 Event Set Pending Register* on page 4-626
0xE001E480	EVENTMASKA	RO	UNKNOWN	*4.13.2 Wake-up Event Mask Registers* on page 4-627
0xE001E484+4n	EVENTMASKn	RO	UNKNOWN	
0xE001E500	ITGU_CTRL	RW	0x00000001	*ITGU and DTGU Control Registers, ITGU_CTRL and DTGU_CTRL* on page 4-591

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-96 IMPLEMENTATION DEFINED register summary (continued)

Address	Name	Type	Reset value	Description
0xE001E504	ITGU_CFG	RO	UNKNOWN	*ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG* on page 4-592
0xE001E510+4n	ITGU_LUTn	• RW if $32n+1<2^{\text{Number of ITGU blocks}}$ • RO if $32n+1\geq 2^{\text{Number of ITGU blocks}}$	0x00000000	*ITGU and DTGU Look Up Table Registers, ITGU_LUTn and DTGU_LUTn* on page 4-593
0xE001E600	DTGU_CTRL	RW	0x00000003	*ITGU and DTGU Control Registers, ITGU_CTRL and DTGU_CTRL* on page 4-591
0xE001E604	DTGU_CFG	RO	UNKNOWN	*ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG* on page 4-592
0xE001E610+4n	DTGU_LUTn	• RW if $32n+1<2^{\text{Number of ITGU blocks}}$ • RO if $32n+1\geq 2^{\text{Number of ITGU blocks}}$	0x00000000	*ITGU and DTGU Look Up Table Registers, ITGU_LUTn and DTGU_LUTn* on page 4-593
0xE001E700	CFGINFOSEL	WO	UNKNOWN	*4.11.1 Processor configuration information selection register, CFGINFOSEL* on page 4-604
0xE001E704	CFGINFORD	RO	UNKNOWN	*4.11.2 Processor configuration information read data register, CFGINFORD* on page 4-607
0xE001ECFC	REVIDR	RO	0x00000000	*4.3.34 Revision ID Register, REVIDR* on page 4-527

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The following registers are reset on Cold reset only. These reset values persist across a system reset or Warm reset.

- *5.6.1 Error Record Feature Register, ERRFR0* on page 5-643.
- *5.6.4 Error Record Miscellaneous Register 10, ERRMISC10* on page 5-648.
- *5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20* on page 5-646.
- *5.6.2 Error Record Primary Status Register, ERRSTATUS0* on page 5-644.
- *5.6.5 Fault Group Status Register, ERRGSR0* on page 5-649.
- *4.10 Implementation defined error banking registers* on page 4-598

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.8 Implementation defined memory system control registers

The implementation defined memory system control registers provide control over memory system implementation and features.

4.8.1 Direct cache access registers

The processor provides a set of registers that allows direct read access to the embedded RAM associated with the *Level 1* (L1) instruction and data cache. Two registers are included for each cache, one to set the required RAM and location, and the other to read out the data.

The following table lists the direct cache access registers.

Table 4-97 Direct cache access registers

Address	Name	Type	Reset value	Description
`0xE001E200`	DCADCRR	RO	UNKNOWN	*Direct Cache Access Read Registers, DCAICRR and DCADCRR* on page 4-582
`0xE001E204`	DCAICRR	RO	UNKNOWN	
`0xE001E210`	DCADCLR	RW	`0x00000000`	*Direct Cache Access Location Registers, DCAICLR and DCADCLR* on page 4-580
`0xE001E214`	DCAICLR	RW	`0x00000000`	

Note

The processor has an external signal, **LOCKDCAIC**, that disable access to the instruction cache direct cache access registers DCAICLR and DCAICRR.

Asserting this signal prevents direct access to the instruction cache Tag or Data RAM content. This is required when using *eXecutable Only Memory* (XOM) on the M-AXI interface

When **LOCKDCAIC** is asserted:
- DCAICLR is RAZ/WI.
- DCAICRR is RAZ.

Direct Cache Access Location Registers, DCAICLR and DCADCLR

The DCAICLR and DCADCLR registers are used by software to set the location to be read from the *Level 1* (L1) instruction cache and data cache respectively.

Usage Constraints

The DCAICLR is RAZ/WI if the L1 instruction cache is not present. The DCADCLR is RAZ/WI if the L1 data cache is not present. If the Security Extension is implemented, these registers are RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

DCAICLR

The following figure shows the DCAICLR bit assignments.

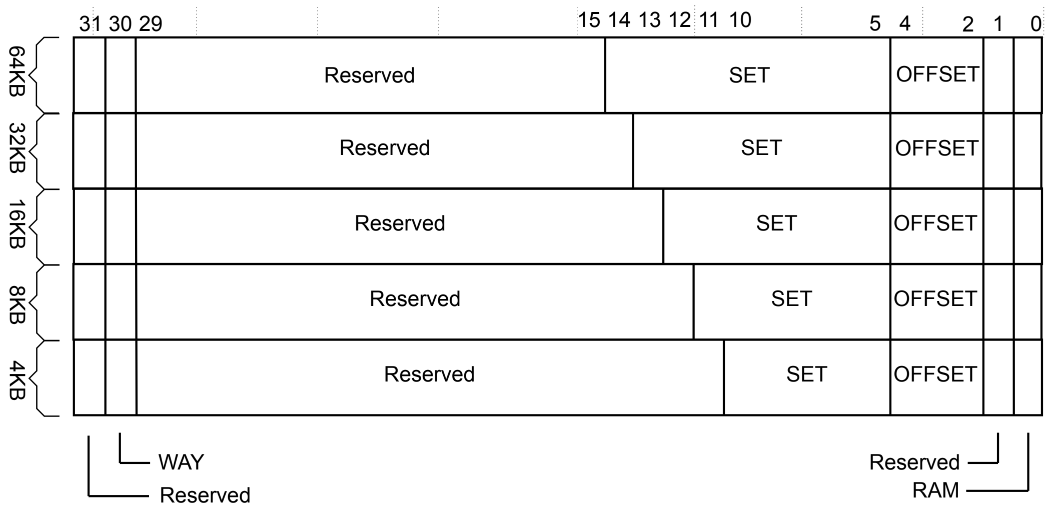


Figure 4-59 DCAICLR bit assignments

The following table shows the DCAICLR bit assignments.

Table 4-98 DCAICLR bit assignments

Bits	Name	Type	Function
[31]	Reserved	-	RES0
[30]	WAY	RO	Cache way
[29:N+1]	Reserved	-	Set index. The value of N depends on the cache size.
[N:5]	SET	RO	The options are: **64KB** N=14 **32KB** N=13 **16KB** N=12 **8KB** N=11 **4KB** N=10
[4:2]	OFFSET	RO	Data offset
[1]	Reserved	-	RES0
[0]	RAMTYPE	RO	RAM type **0** Tag RAM **1** Data RAM

DCADCLR

The following figure shows the DCADCLR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

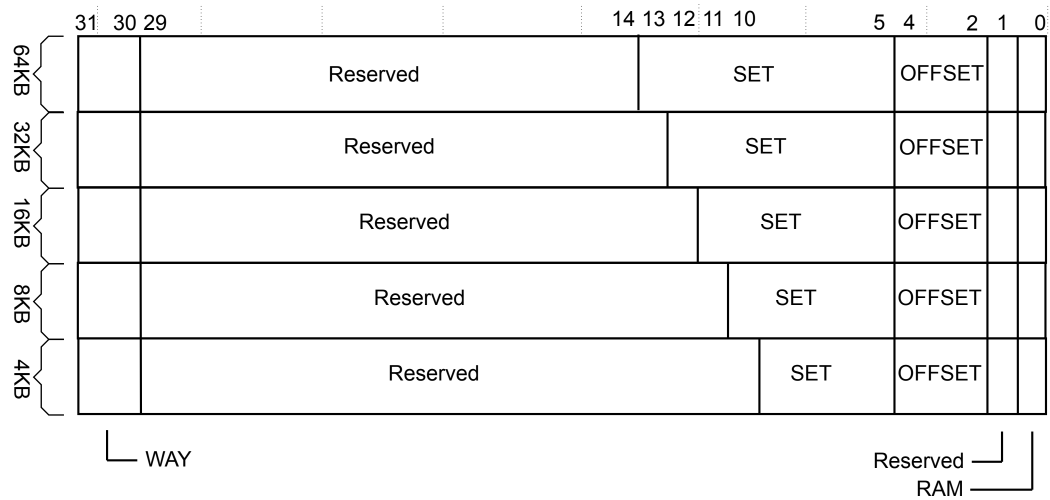


Figure 4-60 DCADCLR bit assignments

The following table shows the DCADCLR bit assignments.

Table 4-99 DCADCLR bit assignments

Bits	Name	Type	Function
[31:30]	WAY	RO	Cache way
[29:N+1]	Reserved	-	Set index. The value of N depends on the cache size.
[N:5]	SET	RO	The options are: **64KB** N=13 **32KB** N=12 **16KB** N=11 **8KB** N=10 **4KB** N=9
[4:2]	OFFSET	RO	Data offset
[1]	Reserved	-	RES0
[0]	RAMTYPE	RO	RAM type **0** Tag RAM **1** Data RAM

Direct Cache Access Read Registers, DCAICRR and DCADCRR

The DCAICRR and DCADCRR registers are used by software to read the data from the *Level 1* (L1) instruction cache and data cache from the location that is determined by the DCAICLR and DCADCLR registers.

Usage Constraints

The DCAICRR is RAZ if the L1 instruction cache is not present. The DCADCRR is RAZ if the L1 data cache is not present. If the Security Extension is implemented, then this register is RAZ from the Non-secure state. Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

These registers are read-only and ignore all writes.
These registers are not banked between Security states.
See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the DCAICRR bit assignments when reading the instruction cache tag RAM.

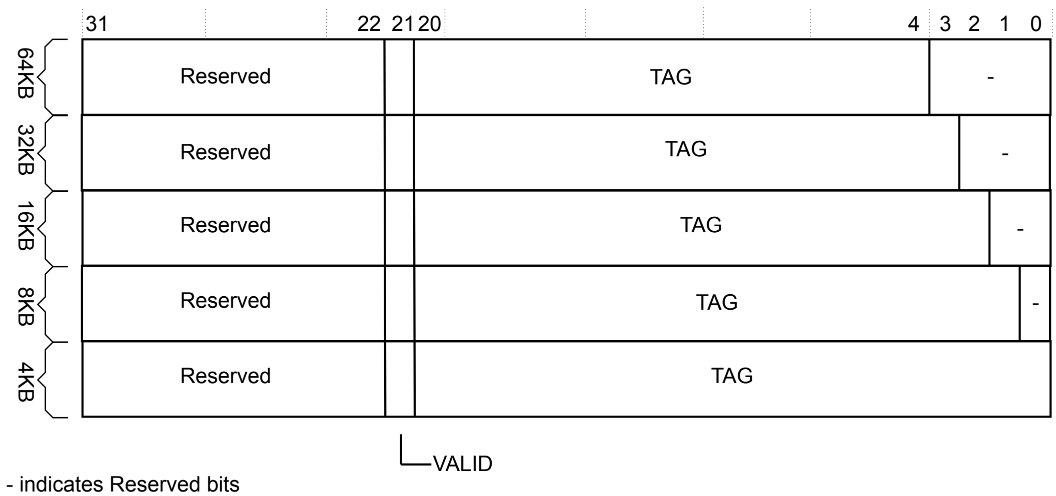


Figure 4-61 DCAICRR bit assignments when reading the instruction cache tag RAM

The following table shows the DCAICRR bit assignments when reading the instruction cache tag RAM.

Table 4-100 DCAICRR bit assignments when reading the instruction cache tag RAM

Bits	Name	Type	Function
[31:22]	-	-	RES0
[21]	VALID	RO	Valid state of the instruction cache line.
[20:N]	TAG	RO	Tag address. The number of significant bits of TAG depends on the instruction cache size. **64KB** N=4 **32KB** N=3 **16KB** N=2 **8KB** N=1 **4KB** N=0
[N-1:0]	-	-	RES0, when N is not 0.

The following figure shows the DCADCRR bit assignments when reading the data cache tag RAM.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

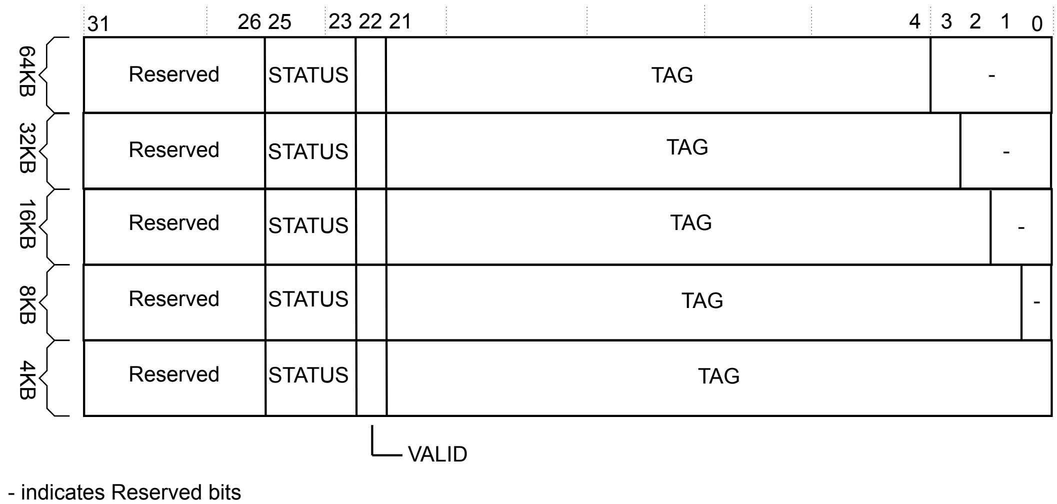


Figure 4-62 DCADCRR bit assignments when reading the data cache tag RAM

The following table shows the DCADCRR bit assignments when reading the data cache tag RAM.

Table 4-101 DCADCRR bit assignments when reading the data cache tag RAM

Bits	Name	Type	Function
[31:26]	Reserved	-	RES0
[25:23]	STATUS	RO	Clean or dirty, transient, and outer attributes of the cache line.
[22]	VALID	RO	Valid state of the data cache line entry.
[21:N]	TAG	RO	Tag address. The number of significant bits of TAG depends on the data cache size. **64KB** N=4 **32KB** N=3 **16KB** N=2 **8KB** N=1 **4KB** N=0
[N-1:0]	-	-	RES0, when N is not 0.

The following figure shows the DCAICRR and DCADCRR bit assignments when reading the instruction or data cache data RAM.

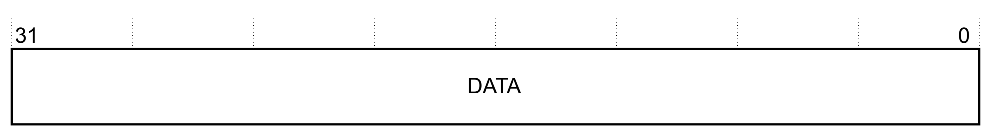


Figure 4-63 DCAICRR and DCADCRR bit assignments when reading the instruction or data cache data RAM

The following table shows the DCAICRR and DCADCRR bit assignments when reading the instruction or data cache data RAM.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-102 DCAICRR and DCADCRR bit assignments when reading the instruction or data cache data RAM

Bits	Name	Type	Function
[31:0]	DATA	RO	Instruction or data cache data entry, ignoring *Error Correcting Code* (ECC).

4.8.2 Memory System Control Register, MSCR

The MSCR controls the memory system features specific to the processor.

Usage constraints

If Security extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.

Configuration

This register is always implemented.
This register is read-only when the data cache is not included.

Attributes

If the Security Extension is included, this register is not banked between Security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the MSCR bit assignments.

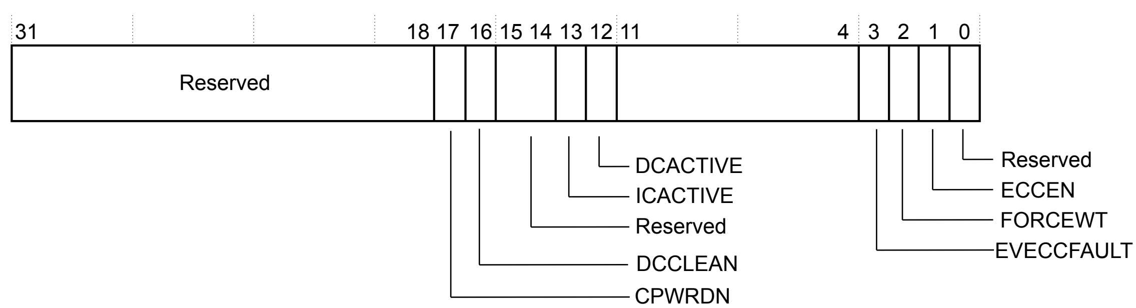


Figure 4-64 MSCR bit assignments

The following table describes the MSCR bit assignments.

Table 4-103 MSCR bit assignments

Bits	Name	Type	Description
[31:17]	Reserved	-	RES0
[16]	DCCLEAN	RW	This bit indicates whether the data cache contains any dirty lines. The options are: **0** *Level 1* (L1) data cache contains at least one dirty line. **1** L1 data cache does not contain any dirty lines. It is cleared to 0 on any write to the L1 data cache that sets the dirty bit. It is cleared to 1 at the end of any automatic L1 data cache invalidate all. Software must only modify this register if it is restoring the state from before the core entered powerdown with the L1 data cache in retention. This field is not updated when a dirty line is evicted, therefore, MCSR.DCCLEAN can be 0, if the cache is currently clean but contained dirty data since the last time it was automatically invalidated. The reset value is 0. If the data cache is not included, this field is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-103 MSCR bit assignments (continued)

Bits	Name	Type	Description
[15:14]	Reserved	-	RES0
[13]	ICACTIVE	RW	This bit indicates whether the L1 instruction cache is active. The options are: **0** L1 instruction cache is inactive. There is no allocation or lookups. Cache maintenance and direct cache access operations are treated as NOPs. **1** L1 instruction cache is active. This implies normal behavior. The reset value is 1. If the L1 instruction cache is not included, this field is RAZ/WI.
[12]	DCACTIVE	RW	This bit indicates whether the L1 data cache is active. The options are: **0** L1 data cache is inactive. There is no allocation or lookups. Cache maintenance and direct cache access operations are treated as NOPs. **1** L1 data cache is active. This implies normal behavior. The reset value is 1. If the L1 data cache is not included, this field is RAZ/WI.
[11:4]	Reserved	-	RES0
[3]	EVECCFAULT	RW	Enables asynchronous BusFault exceptions when data is lost on evictions. The options are: **0** Asynchronous BusFaults are not generated when evicting lines with multi-bit errors in the data. **1** Asynchronous aborts are generated when evicting lines with multi-errors in the data. This is intended for use in systems that do not support the AXI **xPOISON** signals. The reset value is 1. If ECC is not included, this field is RAZ/WI.
[2]	FORCEWT	RW	Enables Forced Write-Through in the L1 data cache. The options are: **0** Force Write-Through is disabled. **1** Force Write-Through is enabled. All Cacheable memory regions are treated as Write-Through. The reset value is 0. If the L1 data cache is not included, this field is RAZ/WI.
[1]	ECCEN	RO	Indicates whether *Error Correcting Code* (ECC) is present and enabled. The options are: **0** ECC not present or not enabled. **1** ECC present and enabled. The reset value depends on the `ECC` Verilog parameter and the external input signal **INITECCEN**. If ECC is not included, this field is RAZ/WI.
[0]	Reserved	-	RES0.

4.8.3 P-AHB Control Register, PAHBCR

The PAHBCR enables accesses to *Peripheral AHB* (P-AHB) interface from software running on the processor. This register also provides information on the range of memory mapped to the interface.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The P-AHB is always memory mapped to a range of the Peripheral and Vendor_SYS regions of the memory map.

Usage Constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from Non-Secure state. Unprivileged access results in a BusFault exception.

Configuration

This register is always implemented.

Attributes

See *4.7 Implementation defined register summary* on page 4-575 for more information.

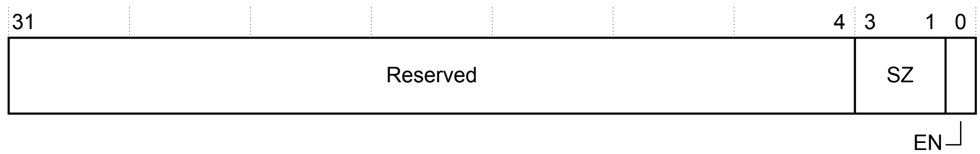


Figure 4-65 PAHBCR bit assignments

The following table shows the PAHBCR bit assignments.

Table 4-104 PAHBCR bit assignments

Bits	Name	Type	Description
[31:4]	-	-	Reserved.
[3:1]	SZ	RO	P-AHB size. The options are: `0b000` 0MB. This implies that P-AHB disabled. `0b001` 64MB. `0b010` 128MB. `0b011` 256MB. `0b100` 512MB. Other encodings are reserved. At reset, the register field is loaded from the **CFGPAHBSZ** input signal. The **CFGPAHBSZ** signal determines the size of the peripheral port memory region.
[0]	EN	RW	P-AHB enable. The options are: **0** P-AHB disabled. When disabled all accesses are made to the M-AXI interface. **1** P-AHB enabled. The reset value is derived from the **INITPAHBEN** signal. This field only affects accesses in the Peripheral region of the memory map. Accesses from the Vendor_SYS region are always enabled.

Note

The processor has an external signal, **LOCKPAHB**, that disables writes to the PAHBCR register from software or from a debug agent connected to the processor. Asserting this signal prevents changes to the PAHBCR.EN bit.

4.8.4 Prefetcher Control Register, PFCR

The PFCR controls the prefetcher. This register can be used to disable the prefetcher if it is causing issues.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage Constraints

If the Security extension is implemented and AIRCR.BFHFNMINS is 0, then this register is RAZ/WI from Non-secure state.
Unprivileged access causes a BusFault exception.

Configuration

This register is always implemented and is RAZ/WI when the L1 data cache is not included.

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PFCR bit assignments.

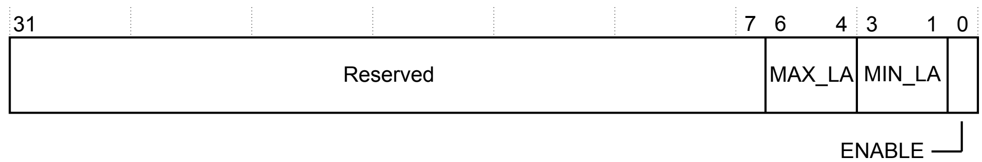


Figure 4-66 PFCR bit assignments

The following table shows the PFCR bit assignments.

Table 4-105 PFCR bit assignments

Bits	Name	Type	Function
[31:7]	Reserved	-	RES0
[6:4]	MAX_LA	RW	Maximum look-ahead distance. If the prefetcher is active, it never has more than MAX_LA number of outstanding linefills. This bit field is not used, and is RES0.
[3:1]	MIN_LA	RW	Minimum look-ahead distance. If the prefetcher is active, it tries to have at least MIN_LA outstanding linefills. This bit field is not used, and is RES0.
[0]	ENABLE	RW	Prefetcher enable. The options are: **0** Prefetcher is disabled. **1** Prefetcher is enabled. The reset value is `0b1`.

4.8.5 TCM Control Registers, ITCMCR and DTCMCR

The ITCMCR and DTCMCR registers enable access to the *Tightly Coupled Memories* (TCMs) by software running on the processor. These registers also provide information on the physical size of the memory connected.

Usage Constraints

If the Security extension is implemented and AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-Secure state. Unprivileged access results in a BusFault exception. If the external input signal, **LOCKTCM** is asserted, these registers are read-only.

Configuration

These registers are always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

If the Security Extension is implemented, these registers are not banked between Security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ITCMCR and DTCMCR bit assignments.

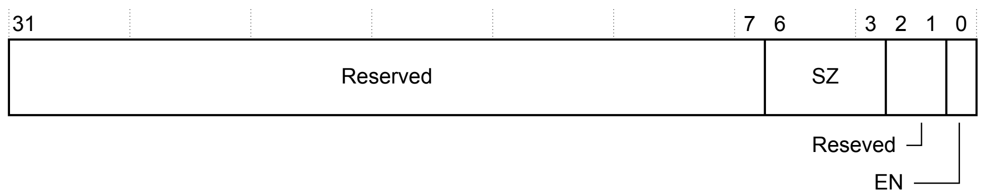


Figure 4-67 ITCMCR and DTCMCR bit assignments

The following table shows the ITCMCR and DTCMCR bit assignments.

Table 4-106 ITCMCR and DTCMCR bit assignments

Bits	Name	Type	Description
[31:7]	-	-	Reserved.
[6:3]	SZ	RO	TCM size indicates the size of the relevant TCM. The options are: `0b0000` No TCM implemented. `0b0011` 4KB. `0b0100` 8KB. `0b0101` 16KB. `0b0110` 32KB. `0b0111` 64KB. `0b1000` 128KB. `0b1001` 256KB. `0b1010` 512KB. `0b1011` 1MB. `0b1100` 2MB. `0b1101` 4MB. `0b1110` 8MB. `0b1111` 16MB. All other encodings are reserved. The reset value is derived from the **CFGITCMSZ** and **CFGDTCMSZ** signals.
[2:1]	Reserved	-	RAZ/WI.
[0]	EN	RW	TCM enable. When a TCM is disabled all accesses are made to the *Master AXI* (M-AXI) interface. The options are: **0** TCM disabled. **1** TCM enabled. The reset value is derived from the **INITTCMEN** pin. This field only affects software accesses to the TCM. Accesses to the TCM from the *S-AHB* interface are always enabled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The processor has external signal, **LOCKTCM**, that disables writes to registers that are associated with the TCM region from software or from a debug agent connected to the processor.

- ITCMCR.
- DTCMCR.

4.8.6 TCM security gate registers

The TCM security gates associated with the *Instruction Tightly Coupled Memory* (ITCM) and *Data Tightly Coupled Memory* (DTCM) are configured using the ITGU_CTRL and DTGU_CTRL registers, respectively. Additionally, there is a set of registers with a group of blocks, ITGU_LUTn and DTGU_LUTn. The configuration of a gate can be read from the read-only ITGU_CFG and DTGU_CFG registers.

The following table lists the TCM security gate registers.

Table 4-107 TCM security gate registers

Address	Name	Type	Reset value	Description
0xE001E500	ITGU_CTRL	RW	0x00000003	*ITGU and DTGU Control Registers, ITGU_CTRL and DTGU_CTRL* on page 4-591
0xE001E504	ITGU_CFG	RO	UNKNOWN	*ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG* on page 4-592
0xE001E510+4n	ITGU_LUTn	• RW if $32n+1<2^{\text{Number of ITGU blocks}}$ • RO if $32n+1\geq2^{\text{Number of ITGU blocks}}$	0x00000000	*ITGU and DTGU Look Up Table Registers, ITGU_LUTn and DTGU_LUTn* on page 4-593
0xE001E600	DTGU_CTRL	RW	0x00000003	*ITGU and DTGU Control Registers, ITGU_CTRL and DTGU_CTRL* on page 4-591
0xE001E604	DTGU_CFG	RO	UNKNOWN	*ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG* on page 4-592
0xE001E610+4n	DTGU_LUTn	• RW if $32n+1<2^{\text{Number of ITGU blocks}}$ • RO if $32n+1\geq2^{\text{Number of ITGU blocks}}$	0x00000000	*ITGU and DTGU Look Up Table Registers, ITGU_LUTn and DTGU_LUTn* on page 4-593

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

The processor has external signals, **LOCKDTGU** and **LOCKITGU**, that disable writes to registers that are associated with the ITCM and DTCM interfaces security gating from software or from a debug agent connected to the processor.

- DTGUCTRL.
- DTGU_LUTn.
- ITGUCTRL.
- ITGU_LUTn.

ITGU and DTGU Control Registers, ITGU_CTRL and DTGU_CTRL

The ITGU_CTRL and DTGU_CTRL registers are the main *TCM Gate Unit* (TGU) control registers for the ITCM and DTCM respectively.

Usage constraints

If the Security Extension is implemented, these registers are RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.
If the Security Extension is not implemented and TCM security gating is not included in the processor, then these registers are RAZ/WI.
If the external input signal **LOCKITGU** is asserted, the ITGU_CTRL register is read-only.
If the external input signal **LOCKDTGU** is asserted, the DTGU_CTRL register is read-only.

Configurations

These registers are always implemented, but their behavior depends on whether the ITGU and DTGU are present.

Attributes

If the Security Extension is implemented, these registers are not banked between security states.
For more information, see *4.7 Implementation defined register summary* on page 4-575.

The following figure shows the ITGU_CTRL and DTGU_CTRL bit assignments.

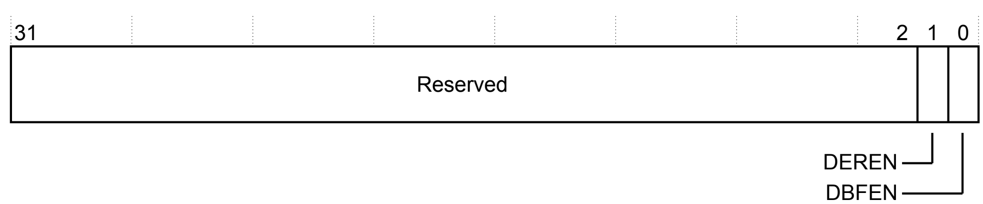


Figure 4-68 ITGU_CTRL and DTGU_CTRL bit assignments

The following table describes the ITGU_CTRL and DTGU_CTRL bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-108 ITGU_CTRL and DTGU_CTRL bit assignments

Field	Name	Type	Description
[31:2]	Reserved	-	-
[1]	DEREN	RW	Enable *Slave AHB* (S-AHB) error response for TGU fault. The options are: **0** Error response not enabled. **1** Error response enabled.
[0]	DBFEN	RW	Enable data side BusFault for TGU fault. The options are: **0** BusFault not enabled. **1** BusFault enabled.

ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG

The ITGU_CFG and DTGU_CFG registers allow the reading of configuration values for the ITGU and DTGU respectively.

Usage constraints

If the Security Extension is implemented, these registers are RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.
If the Security Extension is not implemented and TCM security gating is not included in the processor, then these registers are RAZ/WI.

Configurations

These registers are always implemented, but their behavior depends on whether the ITGU and DTGU are present.

Attributes

If the Security Extension is implemented, these registers are not banked between security states. For more information, see *4.7 Implementation defined register summary* on page 4-575.

The following figure shows the ITGU_CFG and DTGU_CFG bit assignments.

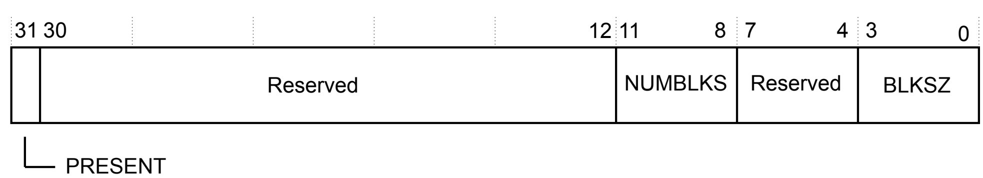


Figure 4-69 ITGU_CFG and DTGU_CFG bit assignments

The following table describes the ITGU_CFG and DTGU_CFG bit assignments.

Table 4-109 ITGU_CFG and DTGU_CFG bit assignments

Field	Name	Type	Description
[31]	PRESENT	-	This field determines if the TGU is present. The options are: **0** TGU not present. **1** TGU is present
[30:12]	Reserved	-	RES0

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-109 ITGU_CFG and DTGU_CFG bit assignments (continued)

Field	Name	Type	Description
[11:8]	NUMBLKS	RO	This is the number of TCM blocks. The value is $2^{NUMBLKS}$. The value of NUMBLKS is: NUMBLKS=(CFGxTCMSZ+4)-xTGUBLKSZ. Where: • NUMBLKS is the number of TCM blocks. • CFGxTCMSZ is the configured TCM size. • xTGUBLKSZ is the configured *Instruction Tightly Coupled Memory Gate Unit* (ITGU) or *Data Tightly Coupled Memory Gate Unit* (DTGU) block size.
[7:4]	Reserved	-	RES0
[3:0]	BLKSZ	RO	TGU block size in bytes. This is $2^{BLKSZ+5}$. This field is determined by the Verilog parameter `xTGUBLKSZ`.

ITGU and DTGU Look Up Table Registers, ITGU_LUTn and DTGU_LUTn

The ITGU_LUTn and DTGU_LUTn registers allow identifying the TGU blocks as being Secure or Non-secure, where n is in the range 0-15.

Usage constraints

If the Security Extension is implemented, these registers are RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.
If the Security Extension is not implemented, then TCM security gating is not included in the processor and these registers are RAZ/WI.
If the external input signal **LOCKITGU** is asserted, the ITGU_LUTn register is read-only.
If the external input signal **LOCKDTGU** is asserted, the DTGU_LUTn register is read-only.

Configurations

The number of programmable blocks depends on the processor configuration and the physical TCM size. This is calculated using the following formula, where x can be I or D for ITGU and DTGU respectively:

$N = 2^{xTGU\_CFG.NUMBLKS}$

Accesses to register fields associated with blocks above the programmable number is treated as RAZ/WI. For more information on the ITGU_CFG and DTGU_CFG registers and the NUMBLKS field, see *ITGU and DTGU Configuration Registers, ITGU_CFG and DTGU_CFG* on page 4-592 .

Attributes

If the Security Extension is implemented, these registers are not banked between security states. For more information, see *4.7 Implementation defined register summary* on page 4-575.

The following figure shows the ITGU_LUTn and DTGU_LUTn bit assignments.

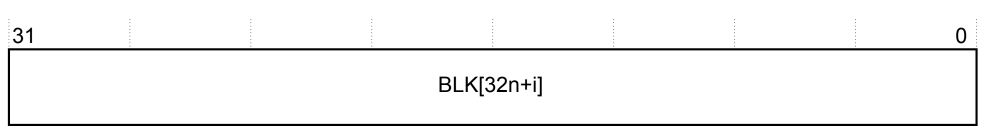


Figure 4-70 ITGU_LUTn and DTGU_LUTn bit assignments

The following table describes the ITGU_LUTn and DTGU_LUTn bit assignments where $0 \leq n \leq 15$, containing the following fields, with the number programmable blocks, $N=2^{xTGU\_CFG.NUMBLKS}$, where x can be I or D for ITGU and DTGU respectively.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-110 ITGU_LUTn and DTGU_LUTn bit assignments for implemented block mapping

Field	Name	Type	Description
[31:0]	BLK[32n+i], where $0 \le i \le 31$	• RW, if $32n+i<N$ • RO, if $32n+i \ge N$	If $32n+i<N$, then the implemented block 32n+i security mapping bit options are: **0** Block mapped as Secure. **1** Block mapped as Non-secure. If $32n+i \ge N$, then the block 32n+i is not implemented and the accesses are treated as RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.9 Implementation defined power mode control

The CPDLPSTATE and DPDLPSTATE registers allow software to control the desired power mode of the functional and debug logic in the processor.

The following table lists the power mode control registers.

Table 4-111 Power mode control registers

Address	Name	Type	Reset value	Description
`0xE001E300`	CPDLPSTATE	RW	`0x00000333`	*4.9.1 Core Power Domain Low Power State Register, CPDLPSTATE* on page 4-595
`0xE001E304`	DPDLPSTATE	RW	`0x00000003`	*4.9.2 Debug Power Domain Low Power State Register, DPDLPSTATE* on page 4-596

4.9.1 Core Power Domain Low Power State Register, CPDLPSTATE

The CPDLPSTATE register specifies the desired low-power states for core (PDCORE), *Extension Processing Unit* (PDEPU), and RAM (PDRAMS) power domains.

Usage Constraints

If AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-secure state.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

This register is not banked between security states.
See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the CPDLPSTATE bit assignments.

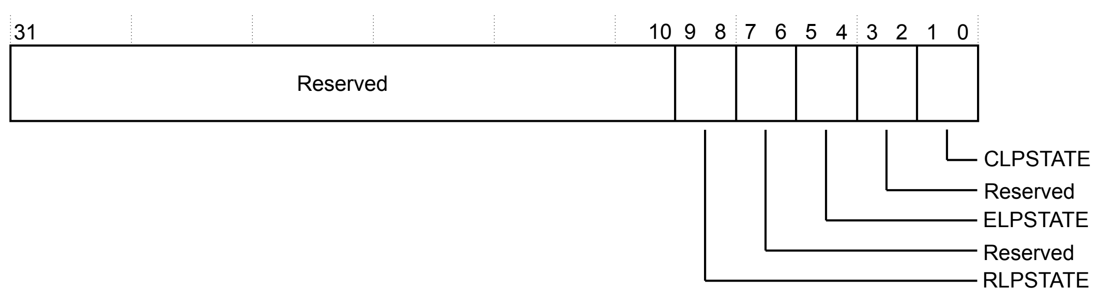


Figure 4-71 CPDLPSTATE bit assignments

The following table shows the CPDLPSTATE bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-112 CPDLPSTATE bit assignments

Bits	Name	Type	Function
[31:10]	Reserved	-	RES0
[9:8]	RLPSTATE	RW	Power-on state for PDRAMS power domain. `0b00` ON. `0b01` Reserved. `0b10` Reserved. `0b11` OFF. **Note** This field is used only to control the Cache/No cache operating mode for the P-Channel. RAM retention is enabled by entering either of the following power modes: • MEM_RET (Cache). • FULL_RET (Cache). • LOGIC_RET (Cache). If the *Level 1* (L1) data cache or instruction cache is not present, this field is RAZ/WI. The reset value is `0b11` on Cold reset.
[7:6]	Reserved	-	RES0
[5:4]	ELPSTATE	RW	Type of low-power state for PDEPU. `0b00` ON. PDEPU is not in low-power state. `0b01` ON, but the clock is off. `0b10` RET. `0b11` OFF. If the *Extension Processing Unit* (EPU) is not present, this field is RAZ/WI. The reset value is `0b11` on Cold reset.
[3:2]	Reserved	-	RES0
[1:0]	CLPSTATE	RW	Type of low-power state for PDCORE. `0b00` ON. PDCORE is not in low-power state. `0b01` ON, but the clock is off. `0b10` RET. `0b11` OFF. The reset value is `0b11` on Cold reset.

4.9.2 Debug Power Domain Low Power State Register, DPDLPSTATE

The DPDLPSTATE register specifies the desired low-power states for the debug (PDDEBUG) power domain.

Usage Constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-secure state.

Configurations

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the DPDLPSTATE bit assignments.

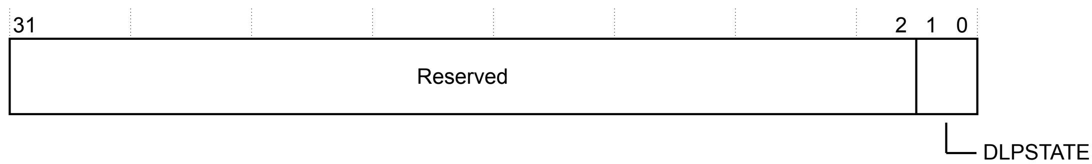


Figure 4-72 DPDLPSTATE bit assignments

The following table shows the DPDLPSTATE bit assignments.

Table 4-113 DPDLPSTATE bit assignments

Bits	Name	Type	Function
[31:2]	Reserved	-	RES0
[1:0]	DLPSTATE	RW	Type of low-power state for PDDEBUG. `0b00` ON. PDDEBUG is not in low-power state. `0b01` ON, but the clock is off. `0b10` RESERVED. Treated as ON, but clock OFF. `0b11` OFF. The reset value is `0b11` at debug Cold reset, which is controlled by the **nDBGRESET** signal.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.10 Implementation defined error banking registers

When the processor is configured to support *Error Correcting Code* (ECC) logic, these registers record errors which occur during memory accesses to the L1 instruction and data cache and the TCM. They also allow certain memory locations to be locked so hard errors can be contained and corrected.

The following table lists the error bank registers.

Table 4-114 Error bank registers

Address	Name	Type	Reset value	Description
`0xE001E100`	IEBR0	RW	`0x00000000`	*4.10.1 Instruction Cache Error Bank Register 0-1, IEBR0 and IEBR1* on page 4-598
`0xE001E104`	IEBR1	RW	`0x00000000`	
`0xE001E110`	DEBR0	RW	`0x00000000`	*4.10.2 Data Cache Error Bank Register 0-1, DEBR0 and DEBR1* on page 4-599
`0xE001E114`	DEBR1	RW	`0x00000000`	
`0xE001E120`	TEBR0	RW	`0x00000000`	*4.10.3 TCM Error Bank Register 0-1, TEBR0 and TEBR1* on page 4-601
`0xE001E124`	TEBRDATA0	RO	`0x00000000`	*Data for TCU Error Bank Register 0-1, TEBRDATA0 and TEBRDATA1* on page 4-602
`0xE001E128`	TEBR1	RW	`0x00000000`	*4.10.3 TCM Error Bank Register 0-1, TEBR0 and TEBR1* on page 4-601
`0xE001E12C`	TEBRDATA1	RO	`0x00000000`	*Data for TCU Error Bank Register 0-1, TEBRDATA0 and TEBRDATA1* on page 4-602

4.10.1 Instruction Cache Error Bank Register 0-1, IEBR0 and IEBR1

The IEBR0 and IEBR1 registers are the two error bank registers that are included for the *Level 1* (L1) instruction cache. These registers are used to record errors that occur during memory accesses to the L1 instruction cache. They also allow certain memory locations to be locked so hard errors can be contained and corrected.

Usage Constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-secure state.
These registers are only reset on Cold reset. Unprivileged access results in a BusFault exception.
These registers are RAZ/WI if the L1 instruction cache is not present or if *Error Correcting Code* (ECC) is excluded.

Configurations

This register is always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

If the Security Extension is implemented, these registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the IEBR0 and IEBR1 bit assignments.

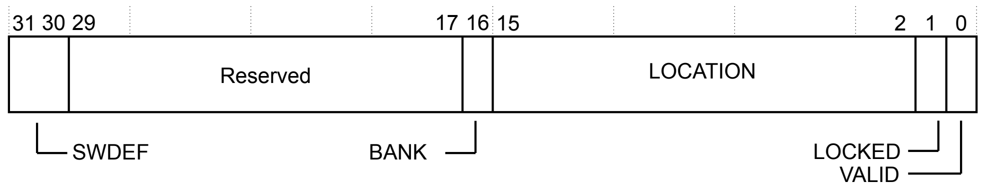


Figure 4-73 IEBR0 and IEBR1 bit assignments

The following table shows the IEBR0 and IEBR1 bit assignments.

Table 4-115 IEBR0 and IEBR1 bit assignments

Bits	Name	Type	Function
[31:30]	SWDEF	RW	User-defined register field. Error detection logic sets this field to `0b00` on a new allocation and on Cold reset.
[29:17]	Reserved	-	RES0
[16]	BANK	RW	Indicates which RAM bank to use. **0** Tag RAM. **1** Data RAM.
[15:2]	LOCATION	RW	Indicates the location in the L1 instruction cache RAM. **[15]** Way **[14:5]** Index **[4:2]** Line word offset.
[1]	LOCKED	RW	Indicates whether the location is locked or not. **0** Location is not locked and available for hardware to allocate. **1** Software has locked the location and hardware is not allowed to allocate to this entry. Only one IEBRn register can be locked at any time. If one of these registers is already locked, then writing to the LOCKED bit of another is ignored. The Cold reset value is 0.
[0]	VALID	RW	Indicates whether the entry is valid or not. **0** Entry is invalid. **1** Entry is valid. The Cold reset value is 0.

4.10.2 Data Cache Error Bank Register 0-1, DEBR0 and DEBR1

The DEBR0 and DEBR1 registers are the two error bank registers that are included for the *Level 1* (L1) data cache. These registers are used to record errors that occur during memory accesses to the L1 data cache. They also allow certain memory locations to be locked so hard errors can be contained and corrected.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage Constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-secure state.

These registers are only reset on Cold reset. Unprivileged access results in a BusFault exception.

These registers are RAZ/WI if the L1 instruction cache is not present or if *Error Correcting Code* (ECC) is excluded.

Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, these registers are not banked between security states.

See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the DEBR0 and DEBR1 bit assignments.

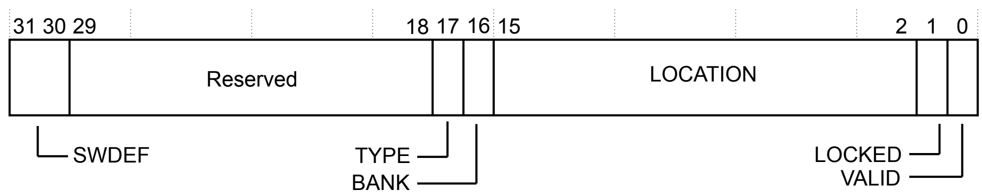


Figure 4-74 DEBR0 and DEBR1 bit assignments

The following table shows the DEBR0 and DEBR1 bit assignments.

Table 4-116 DEBR0 and DEBR1 bit assignments

Bits	Name	Type	Function
[31:30]	SWDEF	RW	User-defined register field. Error detection logic sets this field to `0b00` on a new allocation and on Cold reset.
[29:18]	Reserved	-	RES0
[17]	TYPE	RW	Indicates the error type. **0** Single-bit error. **1** Multi-bit error.
[16]	BANK		Indicates which RAM bank to use. **0** Tag RAM. **1** Data RAM.
[15:2]	LOCATION		Indicates the location in the data cache RAM. **[15:14]** Way. **[13:5]** Index. **[4:2]** Line doubleword offset.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-116 DEBR0 and DEBR1 bit assignments (continued)

Bits	Name	Type	Function
[1]	LOCKED	RW	Indicates whether the location is locked or not. **0** Location is not locked and available for hardware to allocate. **1** Software has locked the location and hardware is not allowed to allocate to this entry. Only one DEBRn register can be locked at any time. If one of these registers is already locked, then writing to the LOCKED bit of another is ignored. The Cold reset value is 0.
[0]	VALID	RW	Indicates whether the entry is valid or not. **0** Entry is invalid. **1** Entry is valid. The Cold reset value is 0.

4.10.3 TCM Error Bank Register 0-1, TEBR0 and TEBR1

The TEBR0 and TEBR1 registers record the location of errors in the TCM.

Usage Constraints

These registers are not banked between security states. If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from Non-secure state, and are only accessible from the Secure state.
These registers are only reset on Cold reset. Unprivileged access results in a BusFault exception.
These registers are RAZ/WI if the L1 instruction cache is not present or if *Error Correcting Code* (ECC) is excluded.

Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, these registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the TEBR0 and TEBR1 bit assignments.

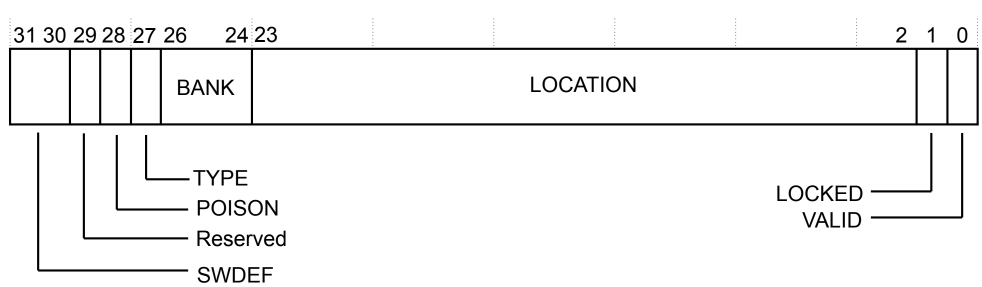


Figure 4-75 TEBR0 and TEBR1 bit assignments

The following table shows the TEBR0 and TEBR1 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-117 TEBR0 and TEBR1 bit assignments

Bits	Name	Type	Function
[31:30]	SWDEF	RW	User-defined register field. Error detection logic sets this field to `0b00` on a new allocation and on Cold reset.
[29]	Reserved	-	RES0
[28]	POISON	RW	Indicates whether a BusFault is generated or not. **0** Load or non-word store to an address that hits this TEBR gets a BusFault. **1** Load or non-word store to an address that hits this TEBR accesses the corresponding TEBRDATA register and does not get a BusFault.
[27]	TYPE	RW	Indicates the error type. **0** Single-bit error. **1** Multi-bit error.
[26:24]	BANK	RW	Indicates which RAM bank to use. `0b000` DTCM0 `0b001` DTCM1 `0b010` DTCM2 `0b011` DTCM3 `0b100` ITCM All other values are RES0.
[23:2]	LOCATION	RW	Indicates the physical location in the data cache RAM.
[1]	LOCKED	RW	Indicates whether the location is locked or not. **0** Location is not locked and available for hardware to allocate. **1** Software has locked the location and hardware is not allowed to allocate to this entry. Only one TEBRn register can be locked at any time. If one of these registers is already locked, then writing to the LOCKED bit of another is ignored. The Cold reset value is 0.
[0]	VALID	RW	Indicates whether the entry is valid or not. **0** Entry is invalid. **1** Entry is valid. If both TEBRn registers are programmed by software with the same LOCATION and BANK field values and VALID is set to 1, then the behavior of TCM accesses is UNPREDICTABLE. The Cold reset value is 0.

Data for TCU Error Bank Register 0-1, TEBRDATA0 and TEBRDATA1

The TEBRDATA0 and TEBRDATA1 registers provide storage for corrected data that is associated with an error.

Usage Constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is 0, then these registers are RAZ/WI from Non-secure state.
These registers are only reset on Cold reset. Unprivileged access results in a BusFault exception.
These registers are RAZ/WI if the L1 instruction cache is not present or if *Error Correcting Code* (ECC) is excluded.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, these registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the TEBRDATA0 and TEBRDATA1 bit assignments.

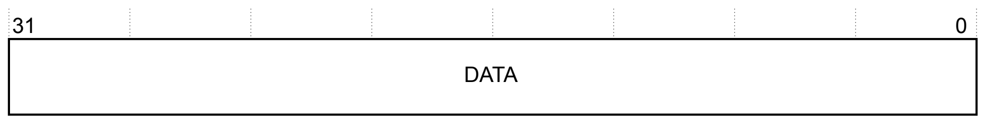


Figure 4-76 TEBRDATA0 and TEBRDATA1 bit assignments

The following table shows the TEBRDATA0 and TEBRDATA1 bit assignments.

Table 4-118 TEBRDATA0 and TEBRDATA1 bit assignments

Bits	Name	Type	Function
[31:0]	DATA	RO	The following access this register instead of the TCM location: • Loads and stores from software running on the processor, if the address matches the location in the corresponding TEBR. • Read and write accesses from the *Slave AHB* (S-AHB).

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.11 Processor configuration information implementation defined registers

The CFGINFOSEL and CFGINFORD registers provide information about the configuration of the processor including the values of all the Verilog parameters used during synthesis and input wire tie-off signals.

The following table lists the processor configuration information registers.

Table 4-119 Processor configuration information registers

Address	Name	Type	Reset value	Description
`0xE001E700`	CFGINFOSEL	WO	UNKNOWN	*4.11.1 Processor configuration information selection register, CFGINFOSEL* on page 4-604
`0xE001E704`	CFGINFORD	RO	UNKNOWN	*4.11.2 Processor configuration information read data register, CFGINFORD* on page 4-607

4.11.1 Processor configuration information selection register, CFGINFOSEL

The CFGINFOSEL register selects the configuration information which can then be read back using CFGINFORD.

Usage constraints
Unprivileged access results in a BusFault exception.

Configurations
This register is always implemented.

Attributes
If the Security Extension is implemented, this register is banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the CFGINFOSEL bit assignments.

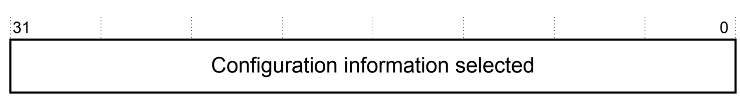


Figure 4-77 CFGINFOSEL bit assignments

The following table describes the CFGINFOSEL bit assignments.

Table 4-120 CFGINFOSEL bit assignments

Field	Name	Type	Description
[31:0]	Configuration information selected	WO	The value of this field depends on the configuration information selected. For more information, see the *Table 4-121 Configuration parameter selection used by the CFGINFOSEL register* on page 4-605.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table lists the CFGINFOSEL register value that depends on the configuration information selected.

Table 4-121 Configuration parameter selection used by the CFGINFOSEL register

CFGINFOSEL value	Configuration information selected
0x1	ICACHESZ
0x2	DCACHESZ
0x3	ECC
0x4	FPU
0x5	MVE
0x6	SECEXT
0x7	CPIF
0x8	MPU_NS
0x9	MPU_S
0xA	SAU
0xB	ITGU
0xC	ITGUBLKSZ
0xD	ITGUMAXBLKS
0xE	DTGU
0xF	DTGUBLKSZ
0x10	DTGUMAXBLKS
0x11	NUMIRQ
0x12	IRQLVL
0x20+n, where 0≤ n≤ 0xF	IRQTIER[(n*32)+31:(n*32)]
0x30+n, where 0≤ n≤ 0xF	IRQDIS[(n*32)+31:(n*32)]
0x40	BUSPROT
0x41	Reserved
0x42	DBGLVL
0x43	ITM
0x44	ETM
0x45	Reserved
0x46	Reserved
0x47	IWIC
0x48	WICLINES
0x49	CTI
0x4A	RAR
0x4B	INITL1RSTDIS
0x4C	CFGMEMALIAS

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

- `INITL1RSTDIS` and `CFGMEMALIAS` select the corresponding external input wire tie-off signal value.
- Input wire tie-off signals also affect the `FPU`, `MVE`, `MPU_NS`, `MPU_S`, and `SAU` values that are read. These signals are **CFGFPU**, **CFGMVE**, **MPUNSDISABLE**, **MPUSDISABLE**, and **SAUDISABLE** respectively. If the input wire tie-off disables the feature, then the configuration indicates that the feature is not supported.
- Parameters `IRQTIER` and `IRQDIS` are selected across multiple values.

CFGINFOSEL register value examples

The following figure shows the CFGINFOSEL bit assignments when `CFGMEMALIAS` parameter is selected.

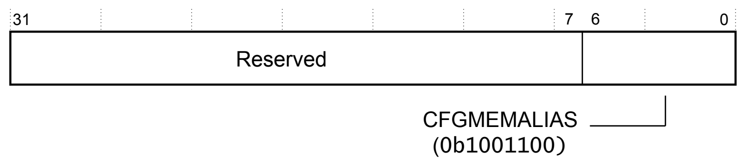


Figure 4-78 CFGINFOSEL bit assignments showing `CFGMEMALIAS`

The following table describes the CFGINFOSEL bit assignments when `CFGMEMALIAS` parameter is selected.

Table 4-122 CFGINFOSEL bit assignments showing `CFGMEMALIAS`

Field	Name	Type	Description
[31:7]	Reserved	-	RES0
[6:0]	`CFGMEMALIAS`	WO	The value is `0x4C`.

The following figure shows the CFGINFOSEL bit assignments when `IRQTIER[63:32]` parameter is selected and n=1.

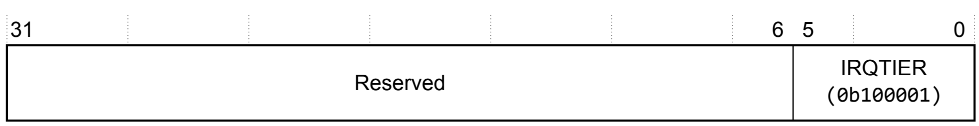


Figure 4-79 CFGINFOSEL bit assignments showing `IRQTIER` when n=1

The following table describes the CFGINFOSEL bit assignments showing `IRQTIER[63:32]` when n=1.

Table 4-123 CFGINFOSEL bit assignments showing `IRQTIER` when n=1

Field	Name	Type	Description
[31:6]	Reserved	-	RES0
[5:0]	`IRQTIER`	WO	The value is `0x21`, indicating `IRQTIER[63:32]`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.11.2 Processor configuration information read data register, CFGINFORD

The CFGINFORD register can be used to display the configuration information that the CFGINFOSEL register selects.

Usage constraints

Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

This register is banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the CFGINFORD bit assignments.

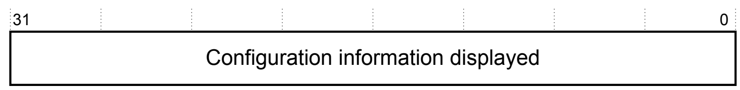


Figure 4-80 CFGINFORD bit assignments

The following table describes the CFGINFORD bit assignments.

Table 4-124 CFGINFORD bit assignments

Field	Name	Type	Description
[31:0]	Configuration information displayed	RO	The value of this field depends on the configuration information selected. For more information, see *4.11.1 Processor configuration information selection register, CFGINFOSEL* on page 4-604.

CFGINFORD register value examples

The following figure shows the CFGINFORD bit assignments when the CFGINFOSEL register selects the `CFGMEMALIAS` parameter.

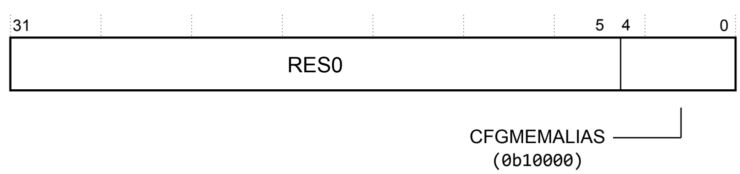


Figure 4-81 CFGINFORD bit assignments showing `CFGMEMALIAS`

The following table describes the CFGINFORD bit assignments when **CFGMEMALIAS** configuration input signal is selected and the alias bit selected is 28.

Table 4-125 CFGINFORD bit assignments showing `CFGMEMALIAS`

Field	Name	Type	Description
[31:5]	Reserved	-	RES0
[4:0]	`CFGMEMALIAS`	RO	The value displayed is `0b10000` to indicate that alias bit 28 has been selected.

The following figure shows the CFGINFORD bit assignments when `IRQTIER` parameter is selected and n=1.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

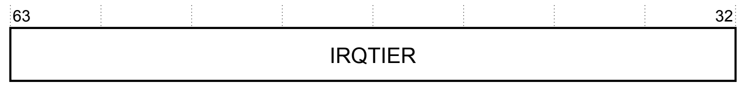


Figure 4-82 CFGINFORD bit assignments showing `IRQTIER` when n=1

The following table describes the CFGINFOSEL bit assignments showing `IRQTIER[63:32]` when n=1. For this example, we are assuming that `IRQTIER[63:32]` is 0 for all interrupts, indicating lowest latency for IRQ32 to IRQ63.

Table 4-126 CFGINFORD bit assignments showing `IRQTIER` when n=1

Field	Name	Type	Description
[63:32]	`IRQTIER`	RO	`0x00000000`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.12 Floating-point and MVE support

The *Extension Processing Unit* (EPU) can be configured to perform floating-point and *M-profile Vector Extension* (MVE) operations.

Scalar floating-point operation

The Cortex-M55 processor can be configured to provide scalar half, single, and double-precision floating-point operation. The floating-point operation is an implementation of the scalar half, single, and double-precision variants of the Floating-point Extension, FPv5 architecture. Configuring the processor to include floating-point supports all half, single, and double-precision data-processing instructions and data types described in the *Arm®v8-M Architecture Reference Manual*.

The processor supports scalar half, single, and double-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. The floating-point functionality that the processor supports also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions.

M-profile Vector Extension operation

The Cortex-M55 processor can be configured to provide Armv8.1-M MVE operation. The MVE functionality that is supported depends on the inclusion of floating-point functionality.

- If floating-point functionality is not included, the processor can be configured to any of the following:
  - Not include MVE.
  - Include the integer subset of MVE only (MVE-I). MVE-I operates on 8-bit, 16-bit, and 32-bit data types.
- If floating-point functionality is included, the processor can be configured to any of the following:
  - Not include MVE.
  - Include the integer subset of MVE only (MVE-I). MVE-I operates on 8-bit, 16-bit, and 32-bit data types.
  - Include the integer, half-precision, and single-precision floating-point MVE (MVE-F). MVE-F operates on half-precision and single-precision floating-point values. MVE-F also includes support for MVE-I.

Vector instructions operate on a fixed vector width of 128 bits.

For more information on the MVE extension, see *Arm®v8-M Architecture Reference Manual*.

Note

- The Cortex-M55 processor provides floating-point computation functionality included with the Armv8.1-M MVE and Floating-point Extension, which is compliant with the *ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic*.
- The scalar Floating-point Extension can be implemented with or without *M-profile Vector Extension - floating-point* (MVE-F).

4.12.1 Floating-point and MVE register summary

The following table shows a summary of the registers in the processor which supports floating-point and *M-profile Vector Extension* (MVE) operations.

These registers are described in the *Arm®v8-M Architecture Reference Manual*. In the following table, SoC designers define the UNKNOWN reset values.

Note

FPCCR, FPCAR, and FPDSCR are banked between security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-127 EPU register summary

Address	Name	Type	Reset value	Description
`0xE000EF34`	FPCCR	RW	`0xC0000004`	*4.12.3 Floating-point Context Control Register, FPCCR* on page 4-611
`0xE000EF38`	FPCAR	RW	`0x00000000`	*4.12.4 Floating-point Context Address Register, FPCAR* on page 4-617
`0xE000EF3C`	FPDSCR	RW	See *4.12.2 FPDSCR and FPSCR register reset values* on page 4-610	*4.12.6 Floating-point Default Status Control Register* on page 4-620
This register is not memory mapped.	FPSCR	RW		*4.12.5 Floating-point Status Control Register, FPSCR* on page 4-618
`0xE000EF40`	MVFR0	RW	*4.12.7 Media and VFP Feature Register 0* on page 4-621	
`0xE000EF44`	MVFR1	RW	*4.12.8 Media and VFP Feature Register 1* on page 4-622	
`0xE000EF48`	MVFR2	RW	*4.12.9 Media and VFP Feature Register 2* on page 4-624	

4.12.2 FPDSCR and FPSCR register reset values

The following table shows the reset values for FPDSCR and FPSCR depending on inclusion and exclusion of floating-point and *M-profile Vector Extension* (MVE) functionality.

Table 4-128 FPDSCR and FPSCR reset values

Register name	Reset value	Floating-point and MVE configuration
FPDSCR	`0x00000000`	Floating-point and MVE are not included.
	`0x00040000`	Scalar half, single, and double-precision floating-point is included. MVE is not included.
		Floating-point is not included. Integer subset of MVE is included.
		Scalar half, single, and double-precision floating-point is included. Integer subset of MVE is included.
		Scalar half, single, and double-precision floating-point is included. Integer and half and single-precision floating-point MVE is included.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-128 FPDSCR and FPSCR reset values (continued)

Register name	Reset value	Floating-point and MVE configuration
FPSCR	`0x00000000`	Floating-point and MVE are not included.
	`0x00040000`	Scalar half, single, and double-precision floating-point is included. MVE is not included.
	UNKNOWN	Floating-point is not included. Integer subset of MVE is included.
	UNKNOWN	Scalar half, single, and double-precision floating-point is included. Integer subset of MVE is included.
	UNKNOWN	Scalar half, single, and double-precision floating-point is included. Integer and half and single-precision floating-point MVE is included.

4.12.3 Floating-point Context Control Register, FPCCR

The FPCCR register sets or returns FPU control data.

Usage Constraints

See *4.12.1 Floating-point and MVE register summary* on page 4-609 for more information.

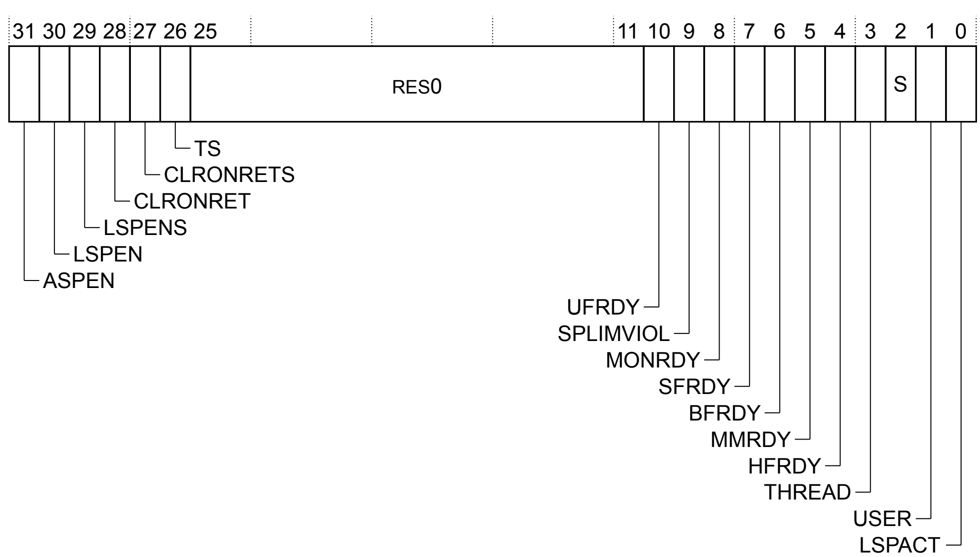
Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, this register is banked on a bit by bit basis. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the FPCCR bit assignments.



Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-129 FPCCR bit assignments without the Security Extension

Bits	Name	Function
[31]	ASPEN	Automatic state preservation enable. Enables CONTROL.FPCA setting on execution of a floating-point instruction. This results in automatic hardware state preservation and restoration, for floating-point context, on exception entry and exit. The possible values of this bit are: **0** Disable CONTROL.FPCA setting on execution of a floating-point instruction. **1** Enable CONTROL.FPCA setting on execution of a floating-point instruction.
[30]	LSPEN	Automatic state preservation enable. Enables lazy context save of floating-point state. The possible values of this bit are: **0** Disable automatic lazy context save. **1** Enable automatic lazy state preservation for floating-point context. Writes to this bit from Non-secure state are ignored if LSPENS is set to one.
[29]	LSPENS	RAZ/WI.
[28]	CLRONRET	Clear on return. Clear floating-point caller saved registers on exception return. The possible values of this bit are: **0** Disabled. **1** Enabled. When set to 1 the caller saved floating-point registers S0 to S15, 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.
[27]	CLRONRETS	RAZ/WI.
[26]	TS	RAZ/WI.
[25:11]	-	Reserved, RES0
[10]	UFRDY	UsageFault ready. Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the UsageFault exception to pending. 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.
[9]	SPLIMVIOL	Stack pointer limit violation. This bit indicates whether the floating-point context violates the stack pointer limit that was active when lazy state preservation was activated. SPLIMVIOL modifies the lazy floating-point state preservation behavior. This bit is banked between Security states. The possible values of this bit are: **0** The existing behavior is retained. **1** The memory accesses associated with the floating-point state preservation are not performed.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-129 FPCCR bit assignments without the Security Extension (continued)

Bits	Name	Function
[8]	MONRDY	DebugMonitor ready. 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 1 in Non-secure state this bit is RAZ/WI.
[7]	SFRDY	RAZ/WI.
[6]	BFRDY	BusFault ready. 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.
[5]	MMRDY	MemManage ready. Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the MemManage exception to pending. 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.
[4]	HFRDY	HardFault ready. Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the HardFault exception to pending. This bit is not banked between Security states. 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.
[3]	THREAD	Thread mode. 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.
[2]	S	RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-129 FPCCR bit assignments without the Security Extension (continued)

Bits	Name	Function
[1]	USER	Indicates the privilege level of the software executing, when the processor allocated the floating point stack. The possible values of this bit are: **0** Privileged level. **1** Unprivileged level.
[0]	LSPACT	Lazy state preservation active. Indicates whether lazy preservation of the floating-point state is active. The possible values of this bit are: **0** Lazy state preservation is not active. **1** Lazy state preservation is active.

Table 4-130 FPCCR bit assignments with the Security Extension

Bits	Name	Function
[31]	ASPEN	Automatic state preservation enable. Enables CONTROL.FPCA setting on execution of a floating-point instruction. This results in automatic hardware state preservation and restoration, for floating-point context, on exception entry and exit. The possible values of this bit are: **0** Disable CONTROL.FPCA setting on execution of a floating-point instruction. **1** Enable CONTROL.FPCA setting on execution of a floating-point instruction. This bit is banked between Security states.
[30]	LSPEN	Automatic state preservation enable. Enables lazy context save of floating-point state. The possible values of this bit are: **0** Disable automatic lazy context save. **1** Enable automatic lazy state preservation for floating-point context. Writes to this bit from Non-secure state are ignored if LSPENS is set to one. This bit is not banked between Security states.
[29]	LSPENS	Lazy state preservation enable Secure only. This bit controls whether the LSPEN bit is writeable from the Non-secure state. The possible values of this bit are: **0** LSPEN is readable and writeable from both Security states. **1** LSPEN is readable from both Security states. Writes to LSPEN are ignored from the Non-secure state. This bit is not banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-130 FPCCR bit assignments with the Security Extension (continued)

Bits	Name	Function
[28]	CLRONRET	Clear on return. Clear floating-point caller saved registers on exception return. The possible values of this bit are: **0** Disabled. **1** Enabled. When set to 1 the caller saved floating-point registers S0 to S15, 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 is not banked between Security states.
[27]	CLRONRETS	Clear on return Secure only. This bit controls whether the CLRONRET bit is writeable from the Non-secure state. The possible values of this bit are: **0** The CLRONRET field is accessibly from both Security states. **1** The Non-secure view of the CLRONRET field is read-only. This bit is RAZ/WI for a Non-secure state. This bit is not banked between Security states.
[26]	TS	Treat as Secure. Treat floating-point registers as Secure enable. The possible values of this bit are: **0** Disabled. **1** Enabled. When set to 0 the floating-point registers are treated as Non-secure even when the core 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. This bit is not banked between Security states.
[25:11]	-	Reserved, RES0
[10]	UFRDY	UsageFault ready. Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the UsageFault exception to pending. 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 is banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-130 FPCCR bit assignments with the Security Extension (continued)

Bits	Name	Function
[9]	SPLIMVIOL	Stack pointer limit violation. This bit indicates whether the floating-point context violates the stack pointer limit that was active when lazy state preservation was activated. SPLIMVIOL modifies the lazy floating-point state preservation behavior. 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. If the floating-point is in Secure state and FPCCR.TS is set to 1 the registers are still zeroed and the floating-point state is lost. This bit is banked between Security states.
[8]	MONRDY	DebugMonitor ready. 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 1 in Non-secure state this bit is RAZ/WI. This bit is not banked between Security states.
[7]	SFRDY	SecureFault ready. If accessed from the Non-secure state, this bit behaves as RAZ/WI. If accessed from the Secure state, this bit 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 not banked between Security states.
[6]	BFRDY	BusFault ready. 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. If in Non-secure state and AIRCR.BFHFNMINS is zero, this bit is RAZ/WI. This bit is not banked between Security states.
[5]	MMRDY	MemManage ready. Indicates whether the software executing when the processor allocated the floating-point stack frame was able to set the MemManage exception to pending. 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 is banked between Security states.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-130 FPCCR bit assignments with the Security Extension (continued)

Bits	Name	Function
[4]	HFRDY	HardFault ready. 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. If in Non-secure state and AIRCR.BFHFNMINS is zero, this bit is RAZ/WI. This bit is not banked between Security states.
[3]	THREAD	Thread mode. Indicates the processor mode when it allocated the floating-point stack frame. 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 is banked between Security states.
[2]	S	Security status of the floating point context. If accessed from the Non-secure state, this bit behaves as RAZ/WI. 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 that the floating-point context belongs to the Non-secure state. **1** Indicates that the floating-point context belongs to the Secure state.
[1]	USER	Indicates the privilege level of the software executing, when the processor allocated the floating point stack. The possible values of this bit are: **0** Privileged level. **1** Unprivileged level. This bit is banked between Security states.
[0]	LSPACT	Lazy state preservation active. Indicates whether lazy preservation of the floating-point state is active. The possible values of this bit are: **0** Lazy state preservation is not active. **1** Lazy state preservation is active. This bit is banked between Security states.

4.12.4 Floating-point Context Address Register, FPCAR

The FPCAR register holds the location of the unpopulated floating-point register space that is allocated on an exception stack frame.

Usage Constraints

See *4.12.1 Floating-point and MVE register summary* on page 4-609 for more information.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

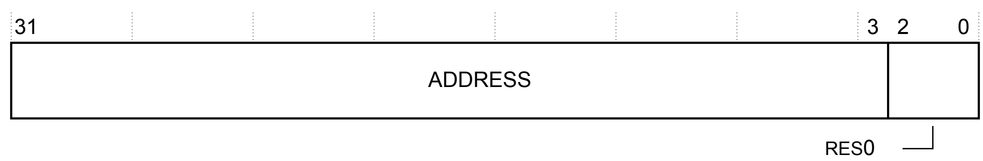
Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, this register is banked on a bit by bit basis. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the FPCAR bit assignments.



The following table describes the FPCAR bit assignments.

Table 4-131 FPCAR bit assignments

Bits	Name	Function
[31:3]	ADDRESS	The location of the unpopulated floating-point register space that is allocated on an exception stack frame.
[2:0]	-	Reserved, RES0

4.12.5 Floating-point Status Control Register, FPSCR

The FPSCR register provides all necessary User level control of the floating-point system.

Usage Constraints

See *4.12.1 Floating-point and MVE register summary* on page 4-609 for more information.

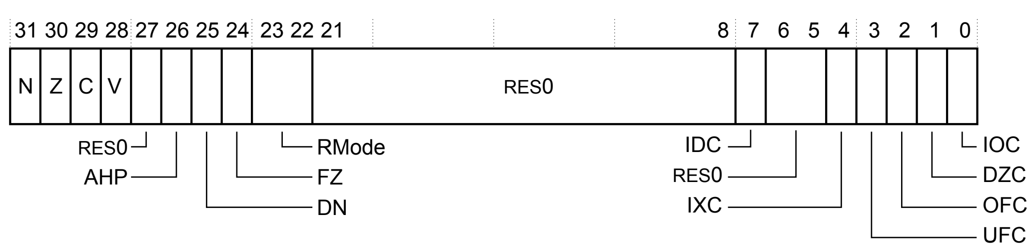
Configurations

This register is always implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the FPSCR bit assignments.



The following table describes the FPSCR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-132 FPSCR bit assignments

Bits	Name	Function
[31]	N	Condition code flags. Floating-point comparison operations update these flags: **N** Negative condition code flag. **Z** Zero condition code flag. **C** Carry condition code flag. **V** Overflow condition code flag.
[30]	Z	
[29]	C	
[28]	V	
[27]	-	Reserved, RES0.
[26]	AHP	Alternative half-precision control bit: **0** IEEE half-precision format selected. **1** Alternative half-precision format selected.
[25]	DN	Default NaN mode control bit: **0** NaN operands propagate through to the output of a floating-point operation. **1** Any operation involving one or more NaNs returns the Default NaN.
[24]	FZ	Flush-to-zero mode control bit: **0** Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard. **1** Flush-to-zero mode enabled.
[23:22]	RMode	Rounding Mode control field. The encoding of this field is: `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. The specified rounding mode is used by almost all floating-point instructions.
[21:20]	Reserved	RES0.
[19]	FZ16	Flush-to-zero mode control for single and double precision Floating-point. This bit determines whether denormal Floating-point values are treated as though they are zero. The possible values of this bit are: **0** Flush-to-zero mode disabled. Behavior of the Floating-point unit is fully compliant with the IEEE754 standard. **1** Flush-to-zero mode enabled. If the Floating-point Extension is not implemented, this bit is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-132 FPSCR bit assignments (continued)

Bits	Name	Function
[18:16]	LTPSIZE	The vector element size used when applying low-overhead-loop tail predication to vector instructions. The possible values of this field are: `0b000` 8 bits `0b001` 16 bits `0b010` 32 bits `0b011` 64 bits `0b100` Tail predication not applied All other values are reserved. The loop hardware behaves as if this field had the value 4 (indicating no low-overhead-loop predication) if no FP context is active. This field reads as 4 and ignores writes if MVE is not implemented. If the Low Overhead Branch Extension is not implemented, this field is RES0.
[15:8]	Reserved	RES0
[7]	IDC	Input Denormal cumulative exception bit, see bits [4:0].
[6:5]	Reserved	RES0
[4]	IXC	Cumulative exception bits for floating-point exceptions, see also bit[7]. Each of these bits is set to 1 to indicate that the corresponding exception has occurred since 0 was last written to it. **IDC, bit[7]** Input Denormal cumulative exception bit. **IXC** Inexact cumulative exception bit. **UFC** Underflow cumulative exception bit. **OFC** Overflow cumulative exception bit. **DZC** Division by Zero cumulative exception bit. **IOC** Invalid Operation cumulative exception bit.
[3]	UFC	
[2]	OFC	
[1]	DZC	
[0]	IOC	

4.12.6 Floating-point Default Status Control Register

The FPDSCR register holds the default values for the floating-point status control data. The processor assigns the floating-point status control data to the FPSCR when it creates a new floating-point context.

Usage Constraints

See *4.12.1 Floating-point and MVE register summary* on page 4-609 for more information.

Configurations

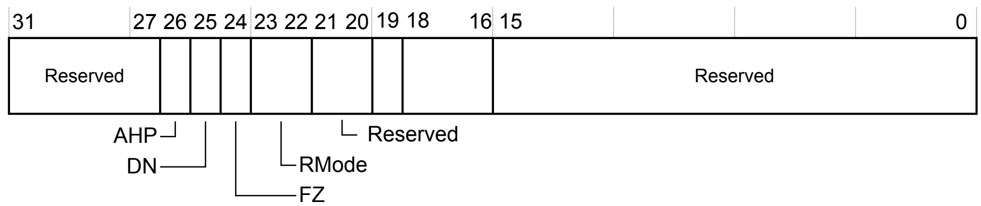
This register is always implemented.

Attributes

If the Security Extension is implemented, this register is banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the FPDSCR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential



The following table describes the FPDSCR bit assignments.

Table 4-133 FPDSCR bit assignments

Bits	Name	Function
[31:27]	Reserved	RES0
[26]	AHP	Default value for FPSCR.AHP
[25]	DN	Default value for FPSCR.DN
[24]	FZ	Default value for FPSCR.FZ
[23:22]	RMode	Default value for FPSCR.RMode
[21:20]	Reserved	RES0
[19]	FZ16	Flush-to-zero mode control bit on half-precision data-processing instructions. Default value for FPSCR.FZ16.
[18:16]	LTPSIZE	The vector element size used when applying low-overhead-loop tail predication to vector instructions. Default value for FPSCR.LTPSIZE. If the Low Overhead Branch Extension is not implemented, this field is RES0.
[15:0]	Reserved	RES0

4.12.7 Media and VFP Feature Register 0

The MVFR0 describes the features provided by the Floating-point and MVE.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is accessible to accesses through unprivileged *Debug AHB* (D-AHB) requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is present if the Floating-point Extension, or *M-profile Vector Extension* (MVE), or both as implemented. If neither are implemented, this register is RES0 .

Attributes

32-bit RO register located at `0xE000EF40`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using MVFR0_NS is located at `0xE002EF40`.
Non-secure alias is provided using MVFR0_NS is located at `0xE002EF40`.

The reset values are:

- When the Floating-point Extension and MVE are not implemented, this register reads `0x00000000`.
- When the Floating-point Extension is not implemented, and only the integer subset of MVE is included, this register reads `0x00000001`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- When half, single, and double-precision Floating-point is included, and MVE is not implemented, this register reads `0x10110221`.
- When half, single, and double-precision Floating-point is included, and the integer subset of MVE is implemented, this register reads `0x10110221`.
- When half, single, and double-precision Floating-point is included, and the integer and half and single-precision floating-point MVE is implemented, this register reads `0x10110221`.

The following figure shows the MVFR0 bit assignments.

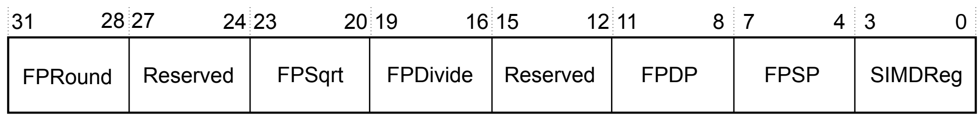


Figure 4-83 MVFR0 bit assignments

The following table describes the MVFR0 bit assignments.

Table 4-134 MVFR0 bit assignments

Bits	Name	Type	Description
[31:28]	FPRound	RO	Floating-point rounding modes. The possible values are: `0b0000` Rounding modes are not supported. `0b0001` All rounding modes are supported.
[27:24]	Reserved	-	RES0
[23:20]	FPSqrt	RO	Floating-point square root. The possible values are: `0b0000` Floating-point square root not supported. `0b0001` Floating-point square root is supported.
[19:16]	FPDivide	RO	Floating-point divide. The possible values are: `0b0000` Floating-point divide operations are not supported. `0b0001` Floating-point divide operations are supported.
[15:12]	Reserved	-	RES0
[11:8]	FPDP	RO	Floating-point double-precision. The possible values are: `0b0000` Floating-point double-precision operations are not supported. `0b0010` Floating-point double-precision operations are supported.
[7:4]	FPSP	RO	Floating-point single-precision. The possible values are: `0b0000` Floating-point single-precision operations are not supported. `0b0010` Floating-point single-precision operations are supported.
[3:0]	SIMDReg	RO	SIMD registers. The value of this field is `0b0001`, indicating 16×64-bit registers.

4.12.8 Media and VFP Feature Register 1

The MVFR0 describes the features provided by the Floating-point and MVE.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is accessible to accesses through unprivileged *Debug AHB* (D-AHB) requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is present if the Floating-point Extension, or *M-profile Vector Extension* (MVE), or both as implemented. If neither are implemented, this register is RES0 .

Attributes

32-bit RO register located at `0xE000EF44`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using MVFR1_NS is located at `0xE002EF44`.
Non-secure alias is provided using MVFR1_NS is located at `0xE002EF44`.

The reset values are:

- When the Floating-point Extension and MVE are not implemented, this register reads `0x00000000`.
- When the Floating-point Extension is not implemented, and only the integer subset of MVE is included, this register reads `0x00000100`.
- When half, single, and double-precision Floating-point is included, and MVE is not implemented, this register reads `0x12100011`.
- When half, single, and double-precision Floating-point is included, and the integer subset of MVE is implemented, this register reads `0x12100111`.
- When half, single, and double-precision Floating-point is included, and the integer and half and single-precision floating-point MVE is implemented, this register reads `0x12100211`.

The following figure shows the MVFR1 bit assignments.

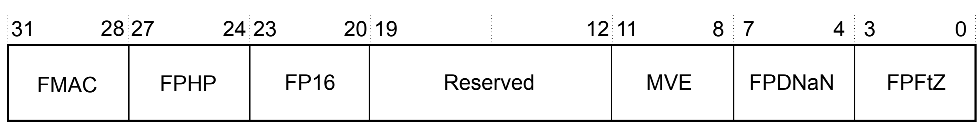


Figure 4-84 MVFR1 bit assignments

The following table describes the MVFR1 bit assignments.

Table 4-135 MVFR1 bit assignments

Bits	Name	Type	Description
[31:28]	FMAC	RO	Fused multiply accumulate instructions. The possible values are: `0b0000` Fused multiply accumulate instructions are not supported. `0b0001` Fused multiply accumulate instructions are supported.
[27:24]	FPHP	RO	Floating-point half-precision conversion. The possible values are: `0b0000` Floating-point half-precision conversion instructions are not supported. `0b0001` Half-precision to single-precision is implemented. `0b0010` Half-precision to single and double-precision are implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-135 MVFR1 bit assignments (continued)

Bits	Name	Type	Description
[23:20]	FP16	RO	Floating-point half-precision data processing. The possible values are: `0b0000` Floating-point half-precision data processing is not supported. `0b0001` Floating-point half-precision data processing is supported.
[19:12]	Reserved	-	RES0
[11:8]	MVE	RO	Indicates support for MVE. The possible values are: `0b0000` MVE is not supported. `0b0001` MVE is supported without Floating-point. `0b0010` MVE is supported with single and double-precision Floating-point.
[7:4]	FPDNaN	RO	Floating-point default NaN. The possible values are: `0b0000` Floating-point default NaN propagation is not supported. `0b0001` Floating-point default NaN propagation is supported.
[3:0]	FPFtZ	RO	Floating-point flush-to-zero. The possible values are: `0b0000` Floating-point flush-to-zero operations not supported. `0b0001` Full denormalized numbers arithmetic supported.

4.12.9 Media and VFP Feature Register 2

The MVFR0 describes the features provided by the Floating-point and MVE.

Usage constraints

Privileged access permitted only. Unprivileged accesses generate a fault.

This register is accessible to accesses through unprivileged *Debug AHB* (D-AHB) requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configuration

This register is present if the Floating-point Extension, or *M-profile Vector Extension* (MVE), or both as implemented. If neither are implemented, this register is RES0 .

Attributes

32-bit RO register located at `0xE000EF48`.
This register is not banked between security states.
If the Security Extension is implemented, this register is not banked between security states.
If the Security Extension is implemented, the Non-secure alias is provided using MVFR2_NS is located at `0xE002EF48`.
Non-secure alias is provided using MVFR2_NS is located at `0xE002EF44`.

The reset values are:

- When the Floating-point Extension and MVE are not implemented, this register reads `0x00000000`.
- When the Floating-point Extension is not implemented, and only the integer subset of MVE is included, this register reads `0x00000000`.
- When half, single, and double-precision Floating-point is included, and MVE is not implemented, this register reads `0x00000040`.
- When half, single, and double-precision Floating-point is included, and the integer subset of MVE is implemented, this register reads `0x00000040`.
- When half, single, and double-precision Floating-point is included, and the integer and half and single-precision floating-point MVE is implemented, this register reads `0x00000040`.

The following figure shows the MVFR2 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

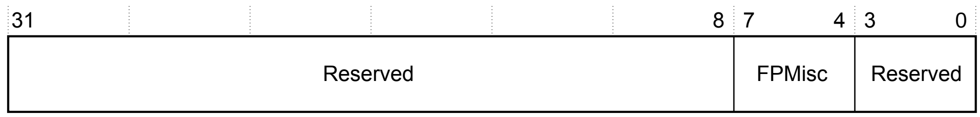


Figure 4-85 MVFR2 bit assignments

The following table describes the MVFR2 bit assignments.

Table 4-136 MVFR2 bit assignments

Bits	Name	Type	Description
[31:8]	Reserved	-	RES0
[7:4]	FPMisc	RO	Floating-point miscellaneous. Indicates support for miscellaneous FP features. The possible values are: `0b0000` Floating-point extensions not implemented. `0b0100` Selection, directed conversion to integer, VMINNM and MAXNM supported.
[3:0]	Reserved	-	RES0

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

4.13 EWIC interrupt status access registers

The *External Wakeup Interrupt Controller* (EWIC) interrupt status access registers, EVENTSPR, EVENTMASKA, and EVENTMASKn registers provide access to the *Nested Vectored Interrupt Controller* (NVIC) state that must be used to carry out software transfers to and from the EWIC in the system for sleep entry and exit when the automatic transfer feature is disabled.

Note

In the following table, SoC designers define the register reset values listed as UNKNOWN.

The following table lists the EWIC interrupt status access registers.

Table 4-137 EWIC interrupt status access registers

Address	Name	Type	Reset value	Description
`0xE001E400`	EVENTSPR	WO	UNKNOWN	*4.13.1 Event Set Pending Register* on page 4-626
`0xE001E480`	EVENTMASKA	RO	UNKNOWN	*4.13.2 Wake-up Event Mask Registers* on page 4-627
`0xE001E484`+4n	EVENTMASKn	RO	UNKNOWN	

4.13.1 Event Set Pending Register

The EVENTSPR is a write-only register that is used to set pending events at wakeup that cannot be directly set in the *Nested Vectored Interrupt Controller* (NVIC) using the architecture programming model.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

If the Security Extension is not implemented, this register is not banked between security states. This register is not banked between security states.
For more information, see *4.7 Implementation defined register summary* on page 4-575. The format of this register is identical to the EWIC_PEND0 register. For more information on the EWIC_PEND0 register, see *A.8 EWIC Pend Event Registers* on page Appx-A-700.

The following figure shows the EVENTPSR bit assignments.

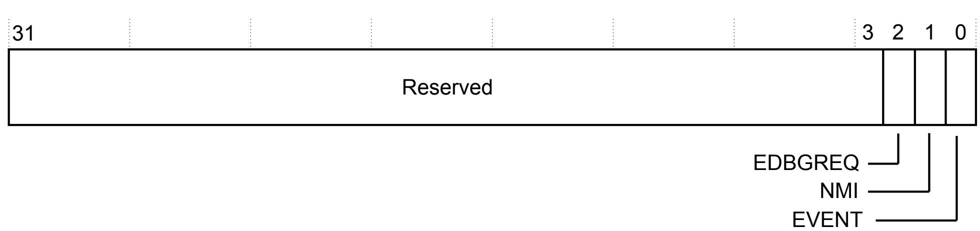


Figure 4-86 EVENTSPR bit assignments

The following table describes the EVENTSPR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 4-138 EVENTSPR bit assignments

Field	Name	Type	Description
[31:3]	Reserved	-	RES0
[2]	EDBGREQ	WO	A write of one to this field causes the processor to behave like an external debug request has occurred. A write of zero is ignored.
[1]	NMI	WO	A write of one to this field causes the processor to behave like a *Non-maskable Interrupt* (NMI) has occurred. A write of zero is ignored.
[0]	EVENT	WO	A write of one to this field causes the processor to behave like an RXEV event has occurred. A write of zero is ignored.

4.13.2 Wake-up Event Mask Registers

The EVENTMASKA and EVENTMASKn are read-only registers that provide the events on sleep entry which cause the processor to wake up. EVENTMASKA includes information about internal events and EVENTMASKn registers cover external interrupt requests (IRQ). There is one register implemented for each of the 32 events that the *Wakeup Interrupt Controller* (WIC) supports. The EVENTMASKA register is always implemented.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state. Unprivileged access results in a BusFault exception.

Configurations

These registers are always implemented.

Attributes

If the Security Extension is not implemented, this register is not banked between security states.
This register is not banked between security states.
See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the EVENTMASKA bit assignments.

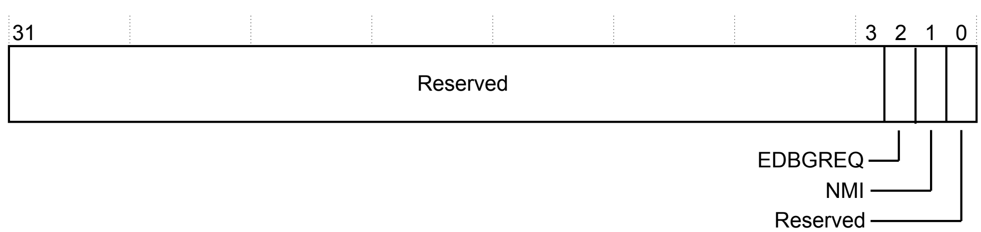


Figure 4-87 EVENTMASKA bit assignments

The following table describes the EVENTMASKA bit assignments.

Table 4-139 EVENTMASKA bit assignments

Field	Name	Type	Description
[31:3]	-	-	Reserved, RES0
[2]	EDBGREQ	RO	Mask for external debug request.
[1]	NMI	RO	Mask for *Non-Maskable Interrupt* (NMI).
[0]	Reserved	-	RES0

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Note

EVENTMASKA[0] is RES0 as the wake-up sensitivity to an external event is determined by the sleep entry instruction and not the processor state. The software transfer sequence must set the EWIC_MASKA.EVENT register field, if the sleep entry instruction is `WFE`.

The following figure shows the EVENTMASKn, where n=0-14, bit assignments.

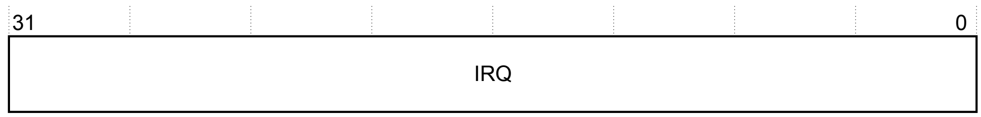


Figure 4-88 EVENTMASKn, where 0≤n≤15, bit assignments.

The following table describes the EVENTMASKn, where n=0-14, bit assignments.

Table 4-140 EVENTMASKn, where 0≤n≤15, bit assignments.

Field	Name	Type	Description
[31:0]	IRQ	RO	Masks for interrupts ((n-1)×32) to (n×32)-1. For EVENTMASKn.IRQ[m] fields, any interrupt that the WIC does not support is RAZ.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 5
Reliability, Availability, and Serviceability Extension support

This chapter describes the *Reliability, Availability, and Serviceability* (RAS) features implemented in the processor.

It contains the following sections:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.1 Cortex®-M55 processor implementation of Arm®v8.1-M RAS

The Cortex-M55 processor implements the Armv8.1-M *Reliability, Availability, and Serviceability* (RAS) features to ensure correct operation in environments where functional safety and high-availability are critical. The Armv8.1-M RAS Extension is always included in the Cortex-M55 processor, however most of the features are only supported when *Error Correcting Code* (ECC) is configured and enabled.

The Cortex-M55 processor standardizes the software interface for fault detection and analysis by supporting the Armv8.1-M RAS Extension. The RAS features supported are *Error Correcting Code* (ECC) for the L1 instruction cache and data cache, and TCMs.

Errors are reported to the system through:
- Output signals on the processor.
- Error bank registers which can be used to mitigate hard errors that cannot be corrected by writing back to the RAM. For more information, see *4.10 Implementation defined error banking registers* on page 4-598.
- The architectural registers defined by the RAS Extension. For more information, see *5.6 Arm®v8.1-M RAS Extension registers* on page 5-643

Supported RAS architectural features

The RAS architecture contains:
- An *Error Synchronization Barrier* (`ESB`) instruction.
- An implicit ESB operation that is inserted after exception entry, exception return, and lazy stacking. This feature is enabled by setting AIRCR.IESB. For more information on AIRCR, see the *Arm®v8-M Architecture Reference Manual*.
- Two ID registers, ERRDEVID and ID_PFR0. For more information on these registers, see the *Arm®v8-M Architecture Reference Manual*.
- A fault status register, RFSR, that is dedicated to RAS events. For more information on:
  — RAS events, see *5.1.1 Cortex®-M55 RAS events* on page 5-630.
  — RFSR, see *5.6.7 Fault Status Register, RFSR* on page 5-650.
- A summary register indicating the nodes that have detected RAS events, ERRGSR. For more information on this register, see *5.6.5 Fault Group Status Register, ERRGSR0* on page 5-649. A node is a unit that can detect RAS events, and for Cortex-M55, a node is the entire processor. Therefore, all RAS events are logged in the same location and the processor supports a single error record.
- Each node has one set of Error Record Registers that can store information about the last RAS event that the node has detected.

  The RAS Error Record Registers are independent of the Error Bank Registers, although they have some common behavior. Either or both of the register types can be used by system software that is handling errors. However, for compatibility across other devices and systems that implement the RAS Extension, the RAS programmers' model must be considered. The RAS Error Record Registers are described in *5.6 Arm®v8.1-M RAS Extension registers* on page 5-643 and the Error Bank Registers are described in *4.10 Implementation defined error banking registers* on page 4-598.

Tip

For a complete description of RAS error types and the information on RAS errors that are produced at the node, see the *Arm® Reliability, Availability, and Serviceability (RAS) Specification*.

This section contains the following subsection:
-

5.1.1 Cortex®-M55 RAS events

The *Reliability, Availability, and Serviceability* (RAS) Extension provides a standard model for recording and reporting errors which occur during the operation of a system.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

In the Cortex-M55 processor, the following are considered as RAS events:

- L1 instruction cache *Error Correcting Code* (ECC) errors.
- L1 data cache ECC errors.
- TCM ECC errors.

Note

For more information on how these RAS events are detected and handled in the Cortex-M55 processor, see *5.2 ECC memory protection behavior* on page 5-632 to get an overview on how instruction cache, data cache, and TCM ECC errors are handled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.2 ECC memory protection behavior

Error Correcting Code (ECC) memory protection is optional. At implementation, you can configure the Cortex-M55 processor to include ECC or not using the Verilog parameter, `ECC`. At Cold reset, if the Cortex-M55 processor is configured with ECC, you can control whether ECC is enabled or not using the static configuration signal **INITECCEN**. **INITECCEN** must only be changed when the processor is powered down and in Cold or Warm reset.

ECC memory protection includes the following protection features:

- Data protection.
- Address decoder protection which involves detecting some of the errors that might occur because of a failure in the address decoder in a RAM instance. A fault in a RAM address decoder circuit might result in the wrong RAM entry being selected, which typically contains data and ECC that are self-consistent. Therefore, address decoder protection is not supported, an ECC error is not generated in this case, but the wrong data is read from the RAM.
- White noise protection which involves protection against faults in the RAM that might also result in no entry being selected, and therefore resulting in reading either all zeros or all ones.

5.2.1 ECC schemes and error type terminology

The Cortex-M55 processor supports two *Error Correcting Code* (ECC) schemes to detect errors.

ECC schemes

SECDED

Single Error Correct Double Error Detect (SECDED) is used on the *Level 1* (L1) data cache and TCM RAMs. The SECDED scheme also provides information on how to correct the error.

DED

Double Error Detect (DED) is used on the L1 instruction cache RAMs. The DED scheme detects single bit and double bit errors. The instruction cache does not need a correction mechanism or scheme because the contents must always be consistent with external memory. Therefore, you can always invalidate the instruction cache RAM to correct its contents.

In the Cortex-M55 processor, the ECC schemes can also support detection of some multi-bit errors where more than two bits are incorrect. Where possible, RAM location information is included in the ECC code to allow detection of faults in the RAM decoder logic.

Error type terminology

The following error type terminology is used in this manual in the context of ECC:

Single-bit error

An error where only one bit of the data or ECC code is incorrect. These errors can usually be corrected.

——— **Note** ———

If an error bit is located in an ECC code field that is associated with the RAM location, this indicates that the read has been carried out from the wrong address in the block. These errors are multi-bit errors because the incorrect line has been read and all of the data is wrong.

———————

Multi-bit error

An error in which more than one bit of data or ECC code is incorrect, or a single-bit error is reported on the part of the address that factors the ECC.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Corrected error (CE)

An ECC error that is detected by hardware and that hardware can correct. These are:

- Single bit errors, which can be corrected inline by flipping the faulty bit.
- All errors which can be corrected by refetching the data from external memory. This includes all instruction cache errors, and all data cache errors when the cache line can be guaranteed to be clean.

For more information on Corrected errors (CEs), see *Arm® Reliability, Availability, and Serviceability (RAS) Specification*.

Uncorrected error (UE)

An ECC error that cannot be corrected or deferred. These are multi-bit errors:

- From the TCMs.
- In an L1 dirty data cache data RAM where it is not guaranteed that the cache line is clean. This includes the case where the ECC indicates that the RAM location is incorrect.
- In an L1 dirty data cache tag RAM where it is not guaranteed that the cache is clean. This includes the case where the ECC indicates that the RAM location is incorrect.

For more information on Uncorrected errors (UEs), see *Arm® Reliability, Availability, and Serviceability (RAS) Specification*.

5.2.2 Enabling ECC

If configured in the processor, *Error Correcting Code* (ECC) is enabled at reset using the input signal **INITECCEN**. Software can determine whether ECC is configured and enabled by reading MSCR.ECCEN.

If ECC is enabled out of reset, the L1 cache must be invalidated before it is enabled to avoid spurious ECC errors being detected because of a mismatch between the data and ECC in the RAM. Automatic instruction and data cache invalidation can be enabled at reset by tying the input signal **INITL1RSTDIS** LOW.

5.2.3 Error detection and processing

The Cortex-M55 processor core is responsible for error detection and processing. The processor core coordinates input from the L1 cache controllers and the TCM controllers. Multiple errors can occur simultaneously, therefore, the processor prioritizes the error processing based on the source.

The following figure shows the prioritization of error processing occurs in the order of decreasing priority.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

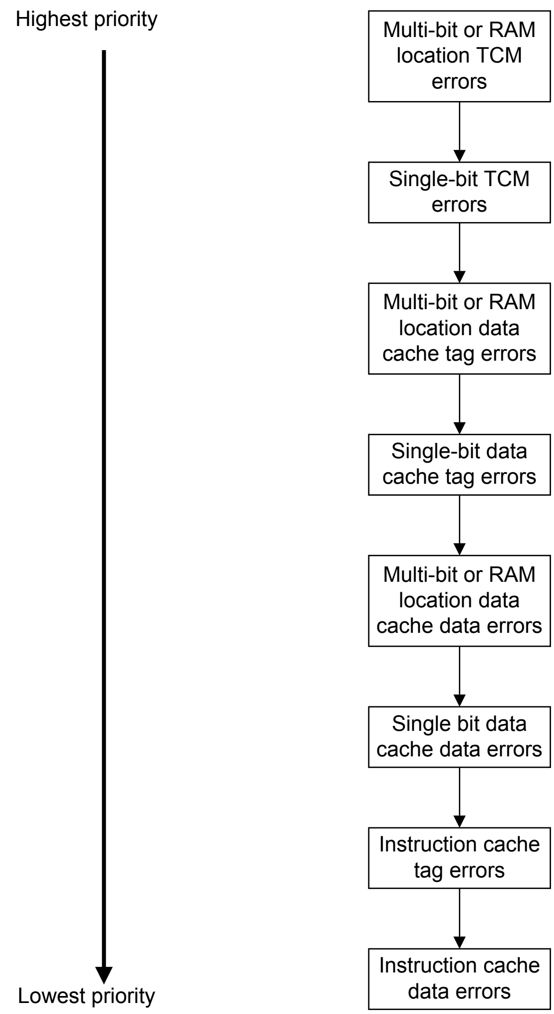


Figure 5-1 Error processing prioritization

The *Data Tightly Coupled Memory* (DTCM) always has higher priority than the *Instruction Tightly Coupled Memory* (ITCM).

Error processing in the L1 data and instruction cache

The cache tag and data RAMs are read during various operations that the processor carries out.

The following table lists these operations.

Table 5-1 Cache RAM access classes

Access type	RAM block read	Notes
Instruction fetch	Instruction tag and data RAM	Two tag banks and up to two data banks
Load request	Data tag and data RAM	4 tag banks and up to four data banks
Dirty line eviction	Data RAM	Entire line is read in parallel
Store buffer address read	Data tag RAM	Four tag banks

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 5-1 Cache RAM access classes (continued)

Access type	RAM block read	Notes
Store buffer data read	Data data RAM	Only used for *Read-Modify-Write* (RMW). RMW is used when the processor writes a partial word when ECC is enabled. Store operations to a cache line, which are less that 64 bits of data must read the data RAM to construct the ECC to write back. This is based on the combination of the current and new data. This read operation can result in an error being detected in the data RAM.
Data cache maintenance	Tag RAM and data RAM	Tag RAM read for address-based and clean operations. Data RAM read for clean evictions.

The error processing operations are:

Instruction fetch
All *Error Correcting Code* (ECC) errors on instruction fetches are processed by invalidating the tag RAM and refetching the line from external memory.

Corrected errors in the L1 data cache for load and store operations
Corrected errors (CE) in the L1 data cache that are detected on load, store, and cache maintenance operations are processed by cleaning (if required) and invalidating the location. For load operations, the data is corrected by replaying, which is refetching and executing the instruction, causing a data cache miss on the invalidated location and reading the correct data from external memory.
Store operations to Write-Allocate memory request a linefill after the error has been processed and then merge the write data into the line as it is allocated to the cache. Store operations to a line in the cache which write less than 64 bits of data must read the data RAM to construct the ECC to write-back, based on a combination of the current and new data. This read operation can result in an error being detected in the data RAM.

Cache maintenance operations

Data cache maintenance operations which operate on an address read all four tag RAMs to check for a match. Instruction cache maintenance operations which operate on an address read two tag RAMs to check for a match. Therefore, they can potentially detect multiple errors unrelated to the requested location. The operation automatically cleans and invalidates all detected errors in sequence. Cache maintenance invalidate by set/way location carried out by Non-secure code always reads the tag because it might contain a dirty line associated with a Secure address, and therefore, it must be cleaned to prevent data loss before being invalidated.

Dirty line eviction
In all cases where a line is evicted, the data RAM associated with the entire line is read out of the cache. Any error detected in this read is corrected inline before being written back to the external memory through the *Master AXI* (M-AXI) interface. If a multi-bit error is detected in the data, the line is marked as poisoned and an imprecise BusFault is raised if MSCR.EVECCFAULT is set.

Multiple errors are processed according to the priority listed in 5.2.3 Error detection and processing on page 5-633 . Errors during load operations are handled by replaying the instruction, therefore, it is possible for errors found in multiple cache ways to not be processed if the original lookup is not repeated. For example, if the replayed load is interrupted.

If data is lost because of a multi-bit ECC error, then an Imprecise BusFault is generated under the following conditions:

- If a data cache eviction is performed, and a multi-bit error is detected in the data RAM and MSCR.EVECCFAULT is set.
- If a data cache line is invalidated because of a multi-bit error detected in the Tag RAM, and MSCR.DCCLEAN is not set.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Although loads do not directly cause BusFaults, they cause ECC maintenance behavior that triggers a BusFault if data is lost. Additionally, if any load sees an ECC error the pipe is flushed, and the load cannot progress until the ECC maintenance has finished. This guarantees that the core does not consume erroneous data until an Imprecise BusFault has been generated.

A multi-bit error on the data cache tag when MSCR.DCCLEAN is asserted is always correctable as the corresponding cache line cannot contain any dirty data.

A multi-bit error on the data cache data when MSCR.EVECCFAULT is deasserted is considered Deferred (DE), because when that line is evicted, it is marked as poisoned. MSCR.EVECCFAULT being deasserted implies that the system supports poisoning.

Any other case of multi-bit errors in the data cache is considered Uncorrected.

Error processing in the TCMs

Error detection and correction are carried out on each of the individual TCMS, that is, ITCM, D0TCM, D1TCM, D2TCM, and D3TCM. Accesses to each of the interfaces are treated in the following way:

- Correctable errors detected during instruction fetch and load operations result in the read being repeated either by refetching the instruction address or replaying the load instruction. The corrected data is written back to the TCM.
- Correctable errors from read requests on the *Slave AHB* (S-AHB) are corrected inline and returned to the system on completion of the transaction.
- Write requests to the TCM with an access size smaller than a complete word or with non-contiguous bytes from S-AHB or *M-profile Vector Extension* (MVE) operations must carry out a *Read-Modify-Write* (RMW) sequence to the TCM. Correctable errors detected during the sequence are corrected inline before the complete store word is written back to the TCM. Uncorrectable errors that are detected on the read phase of an RMW sequence cause the write phase to be abandoned, and the address is marked as poisoned in the error bank register. If the location is read again, a precise BusFault is raised.
- When ECC is enabled, an instruction fetch or load operations might raise a precise BusFault exception, if an *Uncorrected error* (UE) is detected.

5.2.4 Error reporting

Error reporting is done using both registers and output signals.

Corrected errors

Corrected errors (CE) are always transparent to program flow. For more information on Corrected errors (CEs), see *Arm® Reliability, Availability, and Serviceability (RAS) Specification*.

Uncorrected errors

Uncorrected errors (UEs) can result in a precise or imprecise BusFault. If an exception occurs, the source of the error can be determined using the AFSR and RFSR.

An imprecise BusFault is raised when a UE is found in the data cache data RAM during an eviction. If the system supports poisoning, clearing MSCR.EVECCFAULT disables this error. An imprecise BusFault is also raised when a UE is found in the data cache tag RAM and MCSR.DCCLEAN is not set and this type of BusFault cannot be disabled. For more information on Uncorrected errors (UEs), see *Arm® Reliability, Availability, and Serviceability (RAS) Specification*.

Errors detected on accesses to the TCMs never result in an imprecise BusFault.

Errors on the L1 instruction cache, L1 data cache, and TCMs

Errors detected in the L1 instruction cache, L1 data cache, and TCMs are reported on the following external error interface output signals:

- **DMEV0**
- **DMEV1**

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

- **DMEV2**
- **DMEL0[2:0]**
- **DMEL1[2:0]**
- **DMEI0[25:0]**
- **DMEI1[25:0]**

Up to two errors can be reported on the same cycle. If multiple simultaneous errors occur, the priority scheme for reporting is followed. The reporting priority is described in *5.2.3 Error detection and processing* on page 5-633. If up to two errors occur, the location and error class is indicated in **DMELn** and **DMEIn** respectively, and **DMEVn** is asserted. If more than two errors occur, then only information about the two highest priority errors are reported and **DMEV2** is asserted to indicate further information is not available.

Note

A particular ECC error might be reported multiple times on the DME bus.

Error bank registers

The processor includes internal error bank registers which do the following:

- Record the two most recent errors detected.
- Isolate the system from hard errors in the RAM which cannot be corrected by invalidating or overwriting with correct data.

Two error bank registers are included for each source of errors:

- IEBR0 and IEBR1 for the L1 instruction cache.
- DEBR0 and DEBR1 for the L1 data cache.
- TEBR0, TEBR1, TEBRDATA0, and TEBRDATA1 that are shared across the ITCM and DTCM.

Error bank behavior

When an error bank contains a valid entry, any errors detected from the associated RAM address are ignored.

L1 instruction and data cache

For the L1 instruction and data cache, the RAM addresses are masked on a cache lookup and no longer used for allocating a line on a miss, isolating the processor from any potential hard errors in the RAM which could cause incorrect behavior even if corrected data is written from external memory.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

TCMs

For TCMs, each TCM error bank contains a 32-bit data register TEBRDATAn. When a single-bit TCM fault is detected and the error bank is allocated, the corrected data is written to the data register and the TCM memory. Any subsequent read returns the result directly from TEBRDATAn. Writes to an address associated with a valid TCM Error bank is written to both the TEBRDATAn and the TCM RAM to maintain consistency if the error bank is reallocated or cleared by software. If a multi-bit error is detected on a read from the TCM RAM, the error bank TEBRn.POISON field is set. When this field has been set any subsequent read requests to the TCM which matches the error bank address, it will result in an error. A precise BusFault will be raised for a load request from the processor and **HRESP** is asserted on a read on the *Slave AHB* (S-AHB) interface.

Write accesses from store instructions or S-AHB to TCM that match an error bank register with TEBRn.POISON set do not raise a fault. The TEBRn.POISON field is cleared by an aligned 32-bit write to the address associated with the TCM error bank register. The behavior of the poison feature in the TCM error bank register allows hard multi-bit errors to be patched by software. For example:

1. Load from the TCM at an address detects a multi-bit *Error Correcting Code* (ECC) error. TEBRn is allocated, TEBRn.POISON is set, and a fault is raised.
2. Patch write data of 32 bits is stored to the TCM at that address. TEBRDATAn and TCM memory are updated and TEBRn.POISON is cleared.
3. Subsequent read and write transactions to that address are completed as expected.

If this sequence is applied, the failing TCM RAM entry is isolated and normal execution can continue when the write is applied, even when the error is Hard and so cannot be cleared by a patch directly to the RAM. Between steps 1 and 2, read and write transactions with size less than a word continue to raise a fault because the address has not been patched.

The error bank registers are updated when an ECC error from the associated RAM controller has been processed and remains valid until either a subsequent error is detected and processed, or a direct software write to the bank is carried out to clear the data.

Invalid error banks are always allocated in preference to valid error banks. If both error banks contain valid data new errors are allocated using a round-robin approach. Error banks can be locked from being overwritten by writing to the LOCKED field in the error bank register.

The error bank registers are only cleared on Cold reset and retain their content on system reset.

5.2.5 Address decoder and white noise protection

The Cortex-M55 processor includes address decoder and white noise protection.

Address decoder protection

Address decoder protection detects some of the errors that might occur because of a failure in the address decoder in a RAM instance. A fault in a RAM address decoder circuit might result in the wrong RAM entry being selected, which typically, contains data and ECC that are self-consistent. Therefore, an ECC error on the data is not generated in this case, but the wrong data is read from the RAM.

White noise protection

A fault in a RAM might result in no entry being selected, which might result in reading either all zeros or all ones. Protection against such faults is white noise protection.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.3 Interface protection behavior

The Cortex-M55 processor includes parity-based interface protection on the *Master AXI* (M-AXI), *Peripheral AHB* (P-AHB), *External Private Peripheral Bus* (EPPB) master interfaces and the *Slave AHB* (S-AHB) and *Debug AHB* (D-AHB) slave interfaces.

This feature is configured at implementation time by setting the configuration parameter `BUSPROT`. Each interface includes side-channels on the control and data signals providing point-to-point protection between the processor and the interconnect. Odd parity is used to protect signals, with all data and address signals supported on an 8-bit granularity. The interface protection is designed to be used together with other processor and system level features to provide functional safe operation.

Interface protection on AXI is a super-set of the data check feature. **RDATACHK** and **WDATACHK** are considered part of the interface protection signal group. If interface protection is not configured in the processor, **RDATACHK** is unused and **WDATACHK** is tied to 0.

Parity is only checked for each signal on the interface when the signal is valid.

Table 5-2 Parity checking conditions

Interface	Parity checking conditions
M-AXI	For each channel (**AR**, **AW**, **R**, **W**, and **B**): **VALID** and **READY** are always checked. Channel payload signals are checked when **VALID** && **READY**.
P-AHB	**HTRANSP** and **HREADYP** are always checked. **HADDRP**, **HBURSTP**, **HWRITEP**, **HSIZEP**, **HNONSECP**, **HEXCLP**, **HMASTERP**, and **HPROTP** are checked when **HTRANSP**!=IDLE. **HWDATAP** is checked in data phase for write transfer. **HRDATAP** is checked in data phase for read transfer. **HRESPP** and **HEXOKAYP** are checked in data phase.
EPPB	**PSEL** is always checked. **PADDR**, **PPROT**, **PWRITE**, **PENABLE** are checked when **PSEL** == 1. **PREADY** is checked when **PSEL** && **PENABLE**. **PWDATA** and **PSTRB** are checked when **PSEL** && **PWRITE**. **PRDATA** is checked when **PSEL** && **PREADY** && !**PWRITE**. **PSLVERR** is checked when **PSEL** && **PENABLE** && **PREADY**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 5-2 Parity checking conditions (continued)

Interface	Parity checking conditions
S-AHB	**HREADY**, **HREADYOUTS**, **HTRANSS**, and **HSELS** are always checked. **HADDRS**, **HBURSTS**, **HWRITES**, **HSIZES**, **HNONSECS**, and **HPROTS** are checked when **HTRANSS** != IDLE. **HWDATAS** and **HWSTRBS** are checked in data phase for write transfer. **HRDATAS** is checked in data phase for read transfer when **HREADYOUTS** ==1. **HRESPS** is checked in data phase.
D-AHB	**HTRANSD** and **HREADYD** are always checked. **HADDRD**, **HBURSTD**, **HWRITED**, **HSIZED**, **HNONSECD**, and **HPROTD** are checked when **HTRANSD**!=IDLE. **HWDATAD** is checked in data phase for write transfer. **HRDATAD** is checked in data phase for read transfer. **HRESPD** is checked in data phase.

Parity errors detected on the input signals on the interfaces are indicated to the system by a single-cycle pulse on the processor output signal, **DBE**.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.4 M-AXI read access poisoning

The AMBA 5 AXI protocol provides a mechanism to allow devices to indicate that some or all of the read or write data that is transferred is corrupted or poisoned. This information is transferred through **RPOISON** and **WPOISON** signals that are associated with the data read and write channels.

In the Cortex-M55 processor, this feature is used to improve error detection and isolation for uncorrectable errors that are associated with data cache line evictions. If a multi-bit error is detected on a data cache data RAM read during an eviction or cache clean maintenance operation, the **WPOISON** signal is asserted for the associated write data beat on the *Master AXI* (M-AXI).

If poisoning is supported in the system, the information should be stored with the corrupted data. When this address is subsequently read back into the processor, the **RPOISON** signal must be asserted when the data for the beat is presented on M-AXI. Any read burst containing a beat with **RPOISON** asserted is not allocated into the data cache. If a load instruction requests the address associated with **RPOISON** beat, a precise BusFault is raised.

If your system supports data poisoning in this way, then MSCR.EVECCFAULT must be cleared. This reduces the number of imprecise BusFaults generated. If the system does not support data poisoning in this way, then MSCR.EVECCFAULT must be set. This ensures that software is informed (by a BusFault) about all instances of data being lost because of ECC errors. For more information on MSCR, see *4.8.2 Memory System Control Register, MSCR* on page 4-585.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.5 RAS memory barriers

The Armv8.1-M *Reliability, Availability, and Serviceability* (RAS) extension supports the *Error Synchronization Barrier* (`ESB`) instruction.

When this instruction is executed, all outstanding errors which have been detected but not reported are visible to the software running on the system. In the Cortex-M55 processor, this instruction behaves in the same way as the *Data Synchronization Barrier* (`DSB`) instruction. When executed, all outstanding requests in the memory system are completed before the `ESB` instruction completes and any required BusFault exceptions are raised.

The RAS architecture supports another *Error Synchronization Barrier* (ESB) operation, which is implicit, that is, the *Implicit Error Synchronization Barrier* (IESB) operation. This feature is enabled by setting the AIRCR.IESB bit. When enabled, a barrier is inserted after the end of any register stacking or unstacking sequence associated with exception entry, exit, or floating-point register lazy stacking. Execution is halted in the processor until all outstanding transactions, including the stacking sequence have completed and any errors have been reported. The implicit barrier allows software to isolate an error during context switches, with RAS events always being reported in the old context.

Caution

Use IESB carefully because waiting for outstanding transactions to complete on exception entry can increase interrupt latency, particularly if an AXI access associated with the interrupted context takes many cycles to complete. The feature is disabled by default, with AIRCR.IESB set to 0 out of reset.

For more information on AIRCR, see the *Arm®v8-M Architecture Reference Manual*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.6 Arm®v8.1-M RAS Extension registers

The Cortex-M55 processor implements the Armv8.1-M *Reliability, Availability, and Serviceability* (RAS) features to ensure correct operation in environments where functional safety and high-availability are critical. The RAS features can be controlled using the Armv8.1-M RAS Extension registers.

The following table lists the Armv8.1-M RAS Extension registers.

Table 5-3 Armv8.1-M RAS Extension registers

Address	Name	Type	Reset value	Description
`0xE0005000`	ERRFR0	RO	`0x00000101` **Note** `0x00000000`, if the processor is not configured with *Error Correcting Code* (ECC).	*5.6.1 Error Record Feature Register, ERRFR0* on page 5-643
`0xE0005008`	ERRCTRL0	-	-	This register is RES0.
`0xE0005010`	ERRSTATUS0	RW	UNKNOWN	*5.6.2 Error Record Primary Status Register, ERRSTATUS0* on page 5-644
`0xE0005018`	ERRADDR0	RO	UNKNOWN	*5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20* on page 5-646
`0xE000501C`	ERRADDR20	RO	UNKNOWN	*5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20* on page 5-646
`0xE0005020`	ERRMISC00	-	-	This register is RES0.
`0xE0005024`	ERRMISC10	RO	UNKNOWN	*5.6.4 Error Record Miscellaneous Register 10, ERRMISC10* on page 5-648
`0xE0005028`	ERRMISC20	-	-	This register is RES0.
`0xE000502C`	ERRMISC30	-	-	This register is RES0.
`0xE0005030`	ERRMISC40	-	-	This register is RES0.
`0xE0005034`	ERRMISC50	-	-	This register is RES0.
`0xE0005038`	ERRMISC60	-	-	This register is RES0.
`0xE000503C`	ERRMISC70	-	-	This register is RES0.
`0xE0005E00`	ERRGSR0	RO	`0x00000000`	*5.6.5 Fault Group Status Register, ERRGSR0* on page 5-649
`0xE0005FC8`	ERRDEVID	RO	`0x00000001` **Note** `0x00000000`, if the processor is not configured with ECC.	*5.6.6 Error Record Device ID Register, ERRDEVID* on page 5-650
`0xE000EF04`	RFSR	RW	UNKNOWN	*5.6.7 Fault Status Register, RFSR* on page 5-650

5.6.1 Error Record Feature Register, ERRFR0

The ERRFR0 register describes the RAS features that are supported.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRFR0 bit assignments.

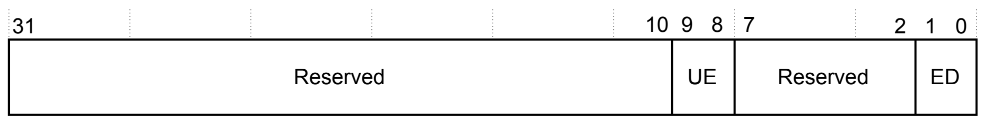


Figure 5-2 ERRFR0 bit assignments

The following table describes the ERRFR0 bit assignments.

Table 5-4 ERRFR0 bit assignments

Field	Name	Type	Description
[31:10]	Reserved	-	RES0
[9:8]	UE	RO	Enable Uncorrected error (UE) reporting as an external abort. `0b01` External abort response for uncorrected errors enabled. This field indicates that uncorrectable errors cause BusFault exceptions.
[7:2]	Reserved	-	RES0
[1:0]	ED	RO	Error reporting and logging. `0b01` Reporting and logging always enabled. This field indicates that logging and reporting of errors cannot be disabled.

5.6.2 Error Record Primary Status Register, ERRSTATUS0

The ERRSTATUS0 register contains information about the *Reliability, Availability, and Serviceability* (RAS) event that is currently logged in record 0.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

The register is not banked between Security states. The read/write behavior depends on the individual fields. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRSTATUS0 bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

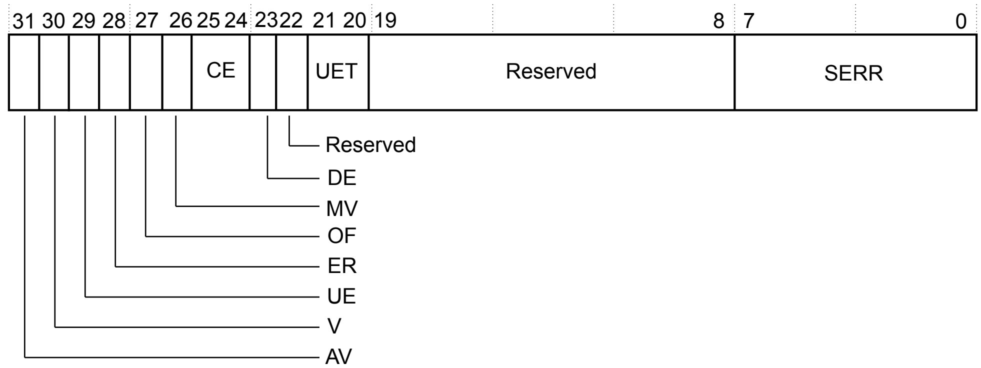


Figure 5-3 ERRSTATUS0 bit assignments

The following table describes the ERRSTATUS0 bit assignments.

Table 5-5 ERRSTATUS0 bit assignments

Field	Name	Type	Description
[31]	AV	RW	Address valid. `0b0` ERRADDR0 is not valid. `0b1` ERRADDR0 is valid. ERRADDR0 is valid only if: • A precise BusFault caused the RAS event. • A TCM *Error Correcting Code* (ECC) error caused the RAS event. This bit is write-one-to-clear.
[30]	V	RW	Status valid. `0b0` ERRSTATUS0 is not valid. `0b1` ERRSTATUS0 is valid. This field is set to 1 on any RAS event. This bit is write-one-to-clear.
[29]	UE	RW	Uncorrected errors (UEs). `0b0` No uncorrectable errors detected. `0b1` At least one uncorrectable error is detected. This bit is write-one-to-clear.
[28]	ER	RW	`0b0` No BusFault caused by RAS event has occurred. `0b1` BusFault caused by RAS event has occurred. This bit is write-one-to-clear.
[27]	OF	RW	`0b0` At most one RAS event has occurred since the last time ERRSTATUS0.V was cleared. `0b1` At least two RAS events have occurred since the last time ERRSTATUS.V was cleared. These events might have occurred at the same time. This bit is write-one-to-clear.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 5-5 ERRSTATUS0 bit assignments (continued)

Field	Name	Type	Description
[26]	MV	RW	Miscellaneous registers valid. `0b0` ERRMISC0 is not valid. `0b1` ERRMISC0 is valid. This field is set to 1 on any RAS event. This bit is write-one-to-clear.
[25:24]	CE	RW	Corrected errors. `0b00` Corrected errors (CEs) have not been detected. `0b10` At least one Corrected error (CE) has been detected. This bit is write-one-to-clear.
[23]	DE	RW	Deferred errors. `0b0` No errors were deferred. `0b1` At least one error was deferred. This bit is write-one-to-clear.
[22]	Reserved	-	RES0.
[21:20]	UET	RW	Uncorrectable error type. `0b01` Uncontainable error, Unrecoverable error (UC). This is for an uncorrectable error that caused an asynchronous BusFault. `0b11` Uncorrectable, Recoverable error (UER). This is for an uncorrectable error that caused a synchronous BusFault. This bit is write-one-to-clear.
[19:8]	Reserved	-	RES0
[7:0]	SERR	RW	Architecturally-defined primary error code. **0** No error. **2** TCM ECC error. **6** L1 data cache or instruction cache data RAM ECC error. **7** L1 data cache or instruction cache tag RAM ECC error. The Cortex-M55 processor does not use the other values of this field.

5.6.3 Error Record Address Registers, ERRADDR0 and ERRADDR20

The ERRADDR0 and ERRADDR20 registers contain information about the address of the *Reliability, Availability, and Serviceability* (RAS) event in record 0.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

These registers are always implemented.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Attributes

These registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRADDR0 bit assignments.

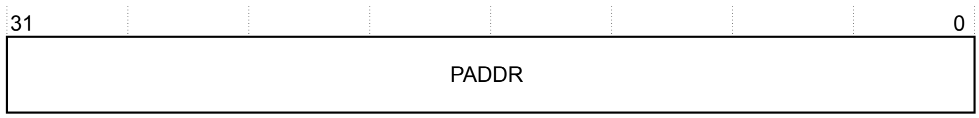


Figure 5-4 ERRADDR0 bit assignments

The following table describes the ERRADDR0 bit assignments.

Table 5-6 ERRADDR0 bit assignments

Field	Name	Type	Description
[31:0]	PADDR	RO	Address of the RAS event. This is the address associated with the memory access that observed *Error Correcting Code* (ECC) error. This field is not valid if ERRADDR20.AI is `0b1`.

The following figure shows the ERRADDR20 bit assignments.

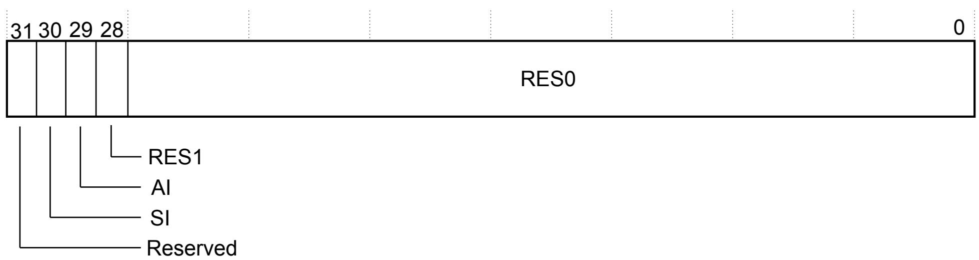


Figure 5-5 ERRADDR20 bit assignments

The following table describes the ERRADDR20 bit assignments.

Table 5-7 ERRADDR20 bit assignments

Field	Name	Type	Description
[31]	Reserved	-	RES0
[30]	SI	RO	Security information incorrect. `0b1` NS bit is not valid. The security information is never guaranteed to be correct.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 5-7 ERRADDR20 bit assignments (continued)

Field	Name	Type	Description
[29]	AI	RO	Address incorrect. `0b0` PADDR is valid. `0b1` PADDR is not valid. PADDR is valid only if: • The RAS event was a precise BusFault. • The RAS event was associated with a TCM ECC error. **Note** If software clears ERRSTATUS.AV, then ERRADDR20.AI is also cleared.
[28]	Reserved	-	RES1
[27:0]	Reserved	-	RES0

5.6.4 Error Record Miscellaneous Register 10, ERRMISC10

The ERRMISC10 register is an IMPLEMENTATION DEFINED error syndrome register for the event in record 0.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

These registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRMISC10 bit assignments.

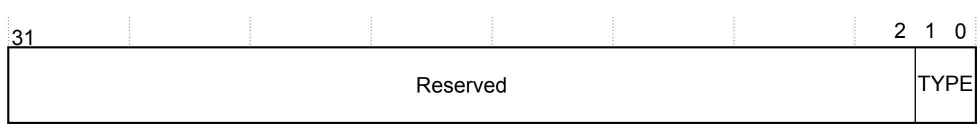


Figure 5-6 ERRMISC10 bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table describes the ERRMISC10 bit assignments.

Table 5-8 ERRMISC10 bit assignments

Field	Name	Type	Description
[31:2]	Reserved	-	RES0
[1:0]	TYPE	RO	Indicates the type of *Reliability, Availability, and Serviceability* (RAS) event logged. `0b00` *Level 1* (L1) instruction cache *Error Correcting Code* (ECC). `0b01` L1 data cache ECC. `0b10` TCM ECC found by load or store executed by the processor. `0b11` TCM ECC found by access from *Slave AHB* (S-AHB).

Note

In the Cortex-M55 processor, only ERRMISC10 is implemented. ERRMISC00 and ERRMISC20-ERRMISC70 are RES0.

5.6.5 Fault Group Status Register, ERRGSR0

The ERRGSR0 register summarizes the valid error records. The Cortex-M55 processor only supports one error record, therefore, only one bit of ERRGSR is active.

Usage constraints

If the Security Extension is implemented and AIRCR.BFHFNMINS is zero, this register is RAZ/WI from the Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

These registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRGSR0 bit assignments.

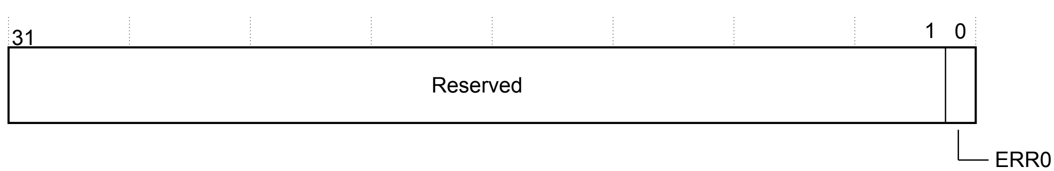


Figure 5-7 ERRGSR0 bit assignments

The following table describes the ERRGSR0 bit assignments.

Table 5-9 ERRGSR0 bit assignments

Field	Name	Type	Description
[31:1]	Reserved	-	RES0
[0]	ERR0	RO	Error record 0 is valid.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

5.6.6 Error Record Device ID Register, ERRDEVID

The ERRDEVID register contains the number of error records that an implementation supports. The Cortex-M55 processor supports a single error record with index 0 if *Error Correcting Code* (ECC) is configured or there are no error records.

Usage constraints

Unprivileged access results in a BusFault exception.
This register is accessible through unprivileged *Debug AHB* (D-AHB) debug requests when either DAUTHCTRL_S.UIDAPEN or DAUTHCTRL_NS.UIDAPEN is set.

Configurations

This register is always implemented.

Attributes

This register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the ERRDEVID bit assignments.

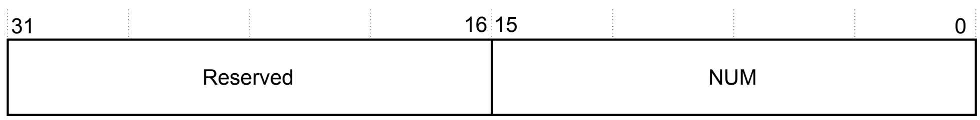


Figure 5-8 ERRDEVID bit assignments

The following table describes the ERRDEVID bit assignments.

Table 5-10 ERRDEVID bit assignments

Field	Name	Type	Description
[31:16]	Reserved	-	RES0
[15:0]	NUM	RO	Maximum Error Record Index+1 `0x0001` If ECC is configured, then one error record with index 0. `0x0000` If ECC is not configured, then there are no error record registers.

5.6.7 Fault Status Register, RFSR

The RFSR reports the fault status of *Reliability, Availability, and Serviceability* (RAS) related faults from *Error Correcting Code* (ECC) errors that are detected in the *Level 1* (L1) instruction cache, data cache, and TCM.

Usage constraints

If the Security extension is implemented and AIRCR.BFHFNMINS is zero, these registers are RAZ/WI from Non-secure state.
If the processor is not configured with ECC, this register is RAZ/WI.
Unprivileged access results in a BusFault exception.

Configurations

This register is always implemented.

Attributes

These registers are not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the RFSR bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

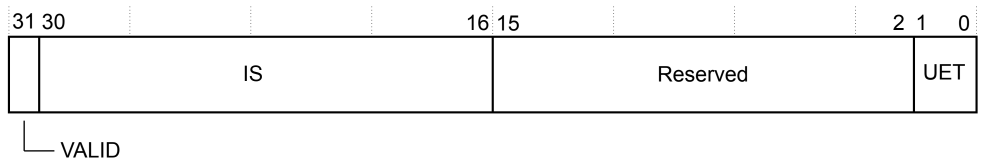


Figure 5-9 RFSR bit assignments

The following table describes the RFSR bit assignments.

Table 5-11 RFSR bit assignments

Bit	Name	Type	Description
[31]	Valid	RW	Indicates whether the register is valid.
[30:16]	IS	RW	Implementation-defined syndrome. Indicates the type of RAS exception that has occurred. `0x0` L1 instruction cache ECC. `0x1` L1 data cache ECC. `0x2` TCM ECC.
[15:2]	Reserved	-	RES0.
[1:0]	UET	RW	Error type. `0b00` Uncontainable error (UC). RAS exception is imprecise. `0b11` Recoverable error (UER). RAS exception is precise. For more information on error types, see the *5.2.1 ECC schemes and error type terminology* on page 5-632.

 Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Chapter 6
Performance Monitoring Unit Extension support

This chapter describes the *Performance Monitoring Unit* (PMU) Extension support.

It contains the following sections:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.1 PMU features

The processor *Data Watchpoint and Trace* (DWT) implements the Armv8.1-M *Performance Monitoring Unit* (PMU). This enables software to get information about events that are taking place in the processor and can be used for performance analysis and system debug.

The PMU supports eight 16-bit event counters and one 32-bit cycle counter. Each event counter can count one event from a list comprising both architectural and implementation defined events. The PMU also supports a chain function which allows the PMU to cascade two of the 16-bit counters into one 32-bit counter. Only odd event counters support the chain feature. PMU counters increment if the appropriate bit in PMU_CNTENSET register is set.

The Armv8.1-M architecture specifies that operation of the PMU counters and DWT profiling counters is mutually exclusive. The processor uses this requirement to share the state used for the counters.

The PMU cycle counter PMU_CCNTR is an alias of the DWT_CYCCNT register. All derived functions of the counter are available whenever either the DWT or the PMU enables the cycle counter. If the DWT is included in the processor, DWT_CTRL.NOCYCCNT is RAZ.

Generating interrupts

If a counter is configured to generate an interrupt when it overflows, DEMCR.MON_PEND is set to 1 to make a Debug Monitor exception pended with DFSR.PMU set to 1. The associated overflow bit programmed by PMU_OVSSET and PMU_OVSCLR indicates which counter triggered the exception. The interrupts are enabled if their corresponding bit programmed by PMU_INTENSET and PMU_INTENCLR is set and DEMCR.MON_EN is 1.

Exporting trace

The PMU can export trace whenever the lower 8 bits of the counters overflow. The PMU issues an event counter packet with the appropriate counter flag set to 1. This occurs on counter increment only, not on software or debugger write. For each counter n, if the lower 8 bits of that counter overflows, the associated OVn bit of the event counter packet is set. If multiple counters overflow during the same period, multiple bits might be set.

The PMU can serve as an event source for the *Cross Trigger Interface* (CTI).

For more information on the registers mentioned in this section, see the *Arm®v8-M Architecture Reference Manual*.

Note

The *Performance Monitoring Unit* (PMU) is included if the *Data Watchpoint and Trace* (DWT) is included in the processor.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.2 PMU events

The following table shows the events that are generated and the numbers that the *Performance Monitoring Unit* (PMU) uses to reference the events.

Table 6-1 PMU events

Event number	Event mnemonic	PMU event bus bit	Event name
0x0000	SW_INCR	0	Instruction architecturally executed, condition code check pass, software increment
0x0001	L1I_CACHE_REFILL	1	L1 instruction cache linefill
0x0003	L1D_CACHE_REFILL	2	L1 data cache linefill
0x0004	L1D_CACHE	3	L1 data cache access
0x0006	LD_RETIRED	4	Instruction architecturally executed, condition code check pass, load
0x0007	ST_RETIRED	5	Instruction architecturally executed, condition code check pass, store
0x0008	INST_RETIRED	6	Instruction architecturally executed.
0x0009	EXC_TAKEN	7	Exception taken.
0x000A	EXC_RETURN	8	Instruction architecturally executed, condition code check pass, exception return.
0x000C	PC_WRITE_RETIRED	9	Instruction architecturally executed, condition code check pass, software change of the PC.
0x000D	BR_IMMED_RETIRED	10	Instruction architecturally executed, immediate branch.
0x000E	BR_RETURN_RETIRED	11	Instruction architecturally executed, condition code check pass, procedure return.
0x000F	UNALIGNED_LDST_RETIRED	12	Instruction architecturally executed, condition code check pass, unaligned load or store.
0x0011	CPU_CYCLES	14	Cycle.
0x0013	MEM_ACCESS	16	Data memory access.
0x0014	L1I_CACHE	17	L1 instruction cache access.
0x0015	L1D_CACHE_WB	18	L1 data cache write-back
0x0019	BUS_ACCESS	19	Any beat access to the M-AXI read interface, M-AXI write interface and any access to P-AHB interface
0x001A	MEMORY_ERROR	20	ECC error for TCMs and caches.
0x001D	BUS_CYCLES	22	Counts the number of cycles on which the M-AXI interface is clocked.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-1 PMU events (continued)

Event number	Event mnemonic	PMU event bus bit	Event name
`0x001E`	CHAIN	23	For an odd-numbered counter, increments when an overflow occurs on the preceding even-numbered counter on the same PE.
`0x0021`	BR_RETIRED	25	Instruction architecturally executed, branch.
`0x0022`	BR_MIS_PRED_RETIRED	26	Instruction architecturally executed, mispredicted branch.
`0x0023`	STALL_FRONTEND	27	If there are no instructions available from the fetch stage of the processor pipeline, the processor considers the front-end of the processor pipeline as being stalled.
`0x0024`	STALL_BACKEND	28	If there is an instruction available from the fetch stage of the pipeline but it cannot be accepted by the decode stage of the processor pipeline, the processor considers the back-end of the processor pipeline as being stalled.
`0x0036`	LL_CACHE_RD	29	L1 data cache read. For the Cortex-M55 processor, this event is the same as L1_CACHE_RD.
`0x0037`	LL_CACHE_MISS_RD	30	L1 data cache read miss. For the Cortex-M55 processor, this event is the same as L1D_CACHE_MISS_RD.
`0x0039`	L1D_CACHE_MISS_RD	31	L1 data cache read miss. For the Cortex-M55 processor, this event is the same as LL_CACHE_MISS_RD.
`0x003C`	STALL	34	No operation sent for execution.
`0x0040`	L1D_CACHE_RD	38	L1 data cache read. For the Cortex-M55 processor, this event is the same as LL_CACHE_RD.
`0x0100`	LE_RETIRED	39	Loop end instruction architecturally executed, entry registered in the LO_BRANCH_INFO cache.
`0x0108`	LE_CANCEL	43	LO_BRANCH_INFO cache containing a valid loop entry cleared while not in the last iteration of the loop.
`0x0114`	SE_CALL_S	45	Call to secure function, resulting in security state change.
`0x0115`	SE_CALL_NS	46	Call to Non-secure function, resulting in security state change
`0x0118`	DWT_CMPMATCH0	47	*Data Watchpoint and Trace* (DWT) comparator 0 match

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-1 PMU events (continued)

Event number	Event mnemonic	PMU event bus bit	Event name
0x0119	DWT_CMPMATCH1	48	DWT comparator 1 match
0x011A	DWT_CMPMATCH2	49	DWT comparator 2 match
0x011B	DWT_CMPMATCH3	50	DWT comparator 3 match
0x0200	MVE_INST_RETIRED	51	*M-profile Vector Extension* (MVE) instruction architecturally executed
0x0204	MVE_FP_RETIRED	53	MVE floating-point instruction architecturally executed.
0x0208	MVE_FP_HP_RETIRED	55	MVE half-precision floating-point instruction architecturally executed
0x020C	MVE_FP_SP_RETIRED	57	MVE single-precision floating-point instruction architecturally executed
0x0214	MVE_FP_MAC_RETIRED	59	MVE floating-point multiply or multiply accumulate instruction architecturally executed
0x0224	MVE_INT_RETIRED	61	MVE integer instruction architecturally executed
0x0228	MVE_INT_MAC_RETIRED	63	MVE integer multiply or multiply-accumulate instruction architecturally executed
0x0238	MVE_LDST_RETIRED	65	MVE load or store instruction architecturally executed
0x023C	MVE_LD_RETIRED	67	MVE load instruction architecturally executed
0x0240	MVE_ST_RETIRED	69	MVE store instruction architecturally executed
0x0244	MVE_LDST_CONTIG_RETIRED	71	MVE contiguous load or store instruction architecturally executed
0x0248	MVE_LD_CONTIG_RETIRED	73	MVE contiguous load instruction architecturally executed
0x024C	MVE_ST_CONTIG_RETIRED	75	MVE contiguous store instruction architecturally executed
0x0250	MVE_LDST_NONCONTIG_RETIRED	77	MVE non-contiguous load or store instruction architecturally executed
0x0254	MVE_LD_NONCONTIG_RETIRED	79	MVE non-contiguous load instruction architecturally executed
0x0258	MVE_ST_NONCONTIG_RETIRED	81	MVE non-contiguous store instruction architecturally executed
0x025C	MVE_LDST_MULTI_RETIRED	83	MVE memory instruction targeting multiple registers architecturally executed
0x0260	MVE_LD_MULTI_RETIRED	85	MVE memory load instruction targeting multiple registers architecturally executed
0x0264	MVE_ST_MULTI_RETIRED	87	MVE memory store instruction targeting multiple registers architecturally executed

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-1 PMU events (continued)

Event number	Event mnemonic	PMU event bus bit	Event name
`0x028C`	MVE_LDST_UNALIGNED_RETIRED	89	MVE unaligned memory load or store instruction architecturally executed
`0x0290`	MVE_LD_UNALIGNED_RETIRED	91	MVE unaligned load instruction architecturally executed
`0x0294`	MVE_ST_UNALIGNED_RETIRED	93	MVE unaligned store instruction architecturally executed
`0x0298`	MVE_LDST_UNALIGNED_NONCONTIG_RETIRED	95	MVE unaligned non-contiguous load or store instruction architecturally executed
`0x02A0`	MVE_VREDUCE_RETIRED	97	MVE vector reduction instruction architecturally executed
`0x02A4`	MVE_VREDUCE_FP_RETIRED	99	MVE floating-point vector reduction instruction architecturally executed
`0x02A8`	MVE_VREDUCE_INT_RETIRED	101	MVE integer vector reduction instruction architecturally executed
`0x02B8`	MVE_PRED	102	Cycles where one or more predicated beats architecturally executed
`0x02CC`	MVE_STALL	103	Stall cycles caused by an MVE instruction
`0x02CD`	MVE_STALL_RESOURCE	104	Stall cycles caused by an MVE instruction because of resource conflicts
`0x02CE`	MVE_STALL_RESOURCE_MEM	105	Stall cycles caused by an MVE instruction because of memory resource conflicts
`0x02CF`	MVE_STALL_RESOURCE_FP	106	Stall cycles caused by an MVE instruction because of floating-point resource conflicts
`0x02D0`	MVE_STALL_RESOURCE_INT	107	Stall cycles caused by an MVE instruction because of integer resource conflicts
`0x02D3`	MVE_STALL_BREAK	108	Stall cycles caused by an MVE chain break
`0x02D4`	MVE_STALL_DEPENDENCY	109	Stall cycles caused by MVE register dependency
`0x4007`	ITCM_ACCESS	110	*Instruction Tightly Coupled Memory* (ITCM) access
`0x4008`	DTCM_ACCESS	111	*Data Tightly Coupled Memory* (ITCM) access
`0x4010`	TRCEXTOUT0	112	*Embedded Trace Macrocell* (ETM) external output 0
`0x4011`	TRCEXTOUT1	113	ETM external output 1
`0x4012`	TRCEXTOUT2	114	ETM external output 2
`0x4013`	TRCEXTOUT3	115	ETM external output 3
`0x4018`	CTI_TRIGOUT4	116	*Cross Trigger Interface* (CTI) output trigger 4
`0x4019`	CTI_TRIGOUT5	117	CTI output trigger 5
`0x401A`	CTI_TRIGOUT6	118	CTI output trigger 6

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-1 PMU events (continued)

Event number	Event mnemonic	PMU event bus bit	Event name
0x401B	CTI_TRIGOUT7	119	CTI output trigger 7
0xC000	ECC_ERR	120	One or more *Error Correcting Code* (ECC) errors detected
0xC001	ECC_ERR_MBIT	121	One or more multi-bit ECC errors detected
0xC010	ECC_ERR_DCACHE	122	One or more ECC errors in the data cache
0xC011	ECC_ERR_ICACHE	123	One or more ECC errors in the instruction cache
0xC012	ECC_ERR_MBIT_DCACHE	124	One or more multi-bit ECC errors in the data cache
0xC013	ECC_ERR_MBIT_ICACHE	125	One or more multi-bit ECC errors in the instruction cache
0xC020	ECC_ERR_DTCM	126	One or more ECC errors in the DTCM
0xC021	ECC_ERR_ITCM	127	One or more ECC errors in the ITCM
0xC022	ECC_ERR_MBIT_DTCM	128	One or more multi-bit ECC errors in the DTCM
0xC023	ECC_ERR_MBIT_ITCM	129	One or more multi-bit ECC errors in the ITCM
0xC100	PF_LINEFILL	130	The prefetcher starts a linefill.
0xC101	PF_CANCEL	131	The prefetcher stops prefetching.
0xC102	PF_DROP_LINEFILL	132	A linefill triggered by the prefetcher has been dropped because of lack of buffering.
0xC200	NWAMODE_ENTER	133	No-write allocate mode entry
0xC201	NWAMODE	134	Write-Allocate store is not allocated into the data cache due to no-write-allocate mode
0xC300	SAHB_ACCESS	135	Read or write access on the S-AHB interface to the TCM
0xC301	PAHB_ACCESS	136	Read or write access to the P-AHB write interface
0xC302	AXI_WRITE_ACCESS	137	Any beat access to M-AXI write interface.
0xC303	AXI_READ_ACCESS	138	Any beat access to M-AXI read interface.
0xC400	DOSTIMEOUT_DOUBLE	140	Denial of Service timeout has fired twice and caused buffers to drain to allow forward progress
0xC401	DOSTIMEOUT_TRIPLE	141	Denial of Service timeout has fired three times and blocked the LSU to force forward progress

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.3 PMU register summary

The following table shows the *Performance Monitoring Unit* (PMU) registers. Each of these registers are 32 bits wide. For more information on these registers, see the *Arm®v8-M Architecture Reference Manual*.

Table 6-2 PMU register summary

Address	Name	Type	Reset value	Description
0xE0003000-0xE000301C	PMU_EVCNTR0-7	RW	0x0000XXXX	*6.4 Performance Monitoring Unit Event Counter Register, PMU_EVCNTR0-7* on page 6-662
0xE000307C	PMU_CCNTR	RW	UNKNOWN	*6.5 Performance Monitoring Unit Cycle Counter Register, PMU_CCNTR* on page 6-663
0xE0003400-0xE000341C	PMU_EVTYPER0-7	RW	0x0000XXXX	*6.6 Performance Monitoring Unit Event Type and Filter Register, PMU_EVTYPER0-7* on page 6-664
0xE000347C	PMU_CCFILTR	-	-	Reserved, RES0.
0xE0003C00	PMU_CNTENSET	RW	0x00000000	*6.7 Performance Monitoring Unit Count Enable Set Register, PMU_CNTENSET* on page 6-665
0xE0003C20	PMU_CNTENCLR	RW	0x00000000	*6.8 Performance Monitoring Unit Count Enable Clear Register, PMU_CNTENCLR* on page 6-666
0xE0003C40	PMU_INTENSET	RW	0x00000000	*6.9 Performance Monitoring Unit Interrupt Enable Set Register, PMU_INTENSET* on page 6-667
0xE0003C60	PMU_INTENCLR	RW	0x00000000	*6.10 Performance Monitoring Unit Interrupt Enable Clear Register, PMU_INTENCLR* on page 6-668
0xE0003C80	PMU_OVSCLR	RW	0x00000000	*6.11 Performance Monitoring Unit Overflow Flag Status Clear Register, PMU_OVSCLR* on page 6-669
0xE0003CA0	PMU_SWINC	WO	0x00000000	*6.13 Performance Monitoring Unit Software Increment Register, PMU_SWINC* on page 6-671
0xE0003CC0	PMU_OVSSET	RW	0x00000000	*6.12 Performance Monitoring Unit Overflow Flag Status Set Register, PMU_OVSSET* on page 6-670
0xE0003E00	PMU_TYPE	RO	0x00A05F08	*6.14 Performance Monitoring Unit Type Register, PMU_TYPE* on page 6-672
0xE0003E04	PMU_CTRL	RW	UNKNOWN	*6.15 Performance Monitoring Unit Control Register, PMU_CTRL* on page 6-673
0xE0003FB8	PMU_AUTHSTATUS	RO	0x00000FEE	*6.16 Performance Monitoring Unit Authentication Status Register, PMU_AUTHSTATUS* on page 6-675
0xE0003FBC	PMU_DEVARCH	RO	0x47700A06	*6.17 Performance Monitoring Unit Device Architecture Register, PMU_DEVARCH* on page 6-678
0xE0003FCC	PMU_DEVTYPE	RO	0x00000016	*6.18 Performance Monitoring Unit Device Type Register, PMU_DEVTYPE* on page 6-679
0xE0003FD0	PMU_PIDR4	RO	0x00000004	*6.23 Performance Monitoring Unit Peripheral Identification Register 4, PMU_PIDR4* on page 6-684
0xE0003FE0	PMU_PIDR0	RO	0x00000022	*6.19 Performance Monitoring Unit Peripheral Identification Register 0, PMU_PIDR0* on page 6-680
0xE0003FE4	PMU_PIDR1	RO	0x000000BD	*6.20 Performance Monitoring Unit Peripheral Identification Register 1, PMU_PIDR1* on page 6-681

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-2 PMU register summary (continued)

Address	Name	Type	Reset value	Description
0xE0003FE8	PMU_PIDR2	RO	0x0000000B	*6.21 Performance Monitoring Unit Peripheral Identification Register 2, PMU_PIDR2* on page 6-682
0xE0003FEC	PMU_PIDR3	RO	0x00000000	*6.22 Performance Monitoring Unit Peripheral Identification Register 3, PMU_PIDR3* on page 6-683
0xE0003FF0	PMU_CIDR0	RO	0x0000000D	*6.24 Performance Monitoring Unit Component Identification Register 0, PMU_CIDR0* on page 6-685
0xE0003FF4	PMU_CIDR1	RO	0x00000090	*6.25 Performance Monitoring Unit Component Identification Register 1, PMU_CIDR1* on page 6-686
0xE0003FF8	PMU_CIDR2	RO	0x00000005	*6.26 Performance Monitoring Unit Component Identification Register 2, PMU_CIDR2* on page 6-687
0xE0003FFC	PMU_CIDR3	RO	0x000000B1	*6.27 Performance Monitoring Unit Component Identification Register 3, PMU_CIDR3* on page 6-688

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.4 Performance Monitoring Unit Event Counter Register, PMU_EVCNTR0-7

The PMU_EVCNTR0-7 registers hold performance counters 0-7, which count events.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

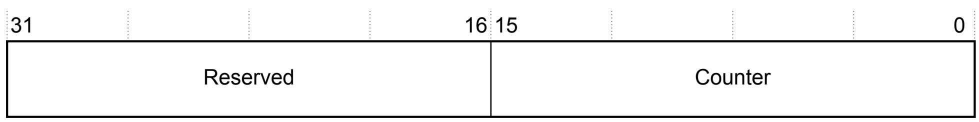
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_EVCNTR0-7 bit assignments.



The following table describes the PMU_EVCNTR0-7 bit assignments.

Table 6-3 PMU_EVCNTR0-7 bit assignments

Bits	Name	Function
[31:16]	-	Reserved, RES0
[15:0]	Counter	Event counter 0-7. The counter counts whenever the selected event occurs, and either of the following conditions are met: • SecureNoninvasiveDebugAllowed()==TRUE. • The source of the event is Non-secure and NoninvasiveDebugAllowed()==TRUE.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.5 Performance Monitoring Unit Cycle Counter Register, PMU_CCNTR

The PMU_CCNTR register holds the value of the cycle counter, which counts processor clock cycles.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

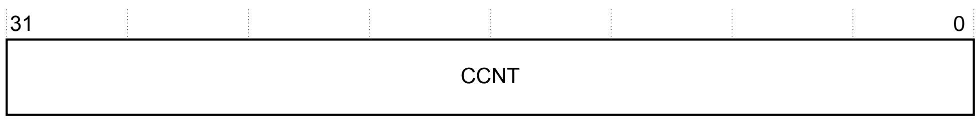
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CCNTR bit assignments.



The following table describes the PMU_CCNTR bit assignments.

Table 6-4 PMU_CCNTR bit assignments

Bits	Name	Function
[31:0]	Cycle count.	The cycle count increments on every processor clock cycle.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.6 Performance Monitoring Unit Event Type and Filter Register, PMU_EVTYPER0-7

The PMU_EVTYPER0-7 registers configure event counters 0-7.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

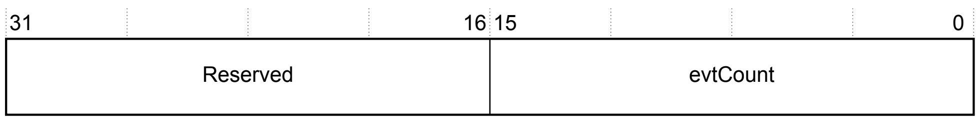
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_EVCNTR0-7 bit assignments.



The following table describes the PMU_EVCNTR0-7 bit assignments.

Table 6-5 PMU_EVCNTR0-7 bit assignments

Bits	Name	Function
[31:16]	-	Reserved, RES0
[15:0]	evtCount	Event to count. The event number of the event that is counted by event counter PMU_EVCNTR0-7. If the associated counter does not support the event number that is written to this register, the value that is read back is UNKNOWN.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.7 Performance Monitoring Unit Count Enable Set Register, PMU_CNTENSET

The PMU_CNTENSET register enables the PMU_CCNTR register and PMU_EVCNTR0-7 registers. Reading this register shows which counters are enabled.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

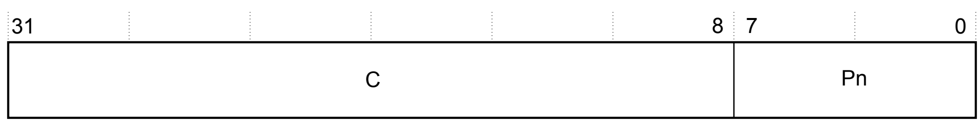
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CNTENSET bit assignments.



The following table describes the PMU_CNTENSET bit assignments.

Table 6-6 PMU_CNTENSET bit assignments

Bits	Name	Function
[31:8]	C	PMU_CCNTR enable bit. The possible values are: **0** When this bit is read with this value, the cycle counter is disabled. When this bit is written to, there is no effect. **1** When this bit is read with this value, the cycle counter is enabled. When written, enables the cycle counter.
[7:0]	Pn	Event counter PMU_EVCNTR0-7 enable bit. The possible values are: `0b0` When this bit is read with this value, this implies that PMU_EVCNTRn is disabled. When this bit is written to, there is no effect. `0b1` When this bit is read with this value, this implies that PMU_EVCNTRn is enabled. When this bit is written to, PMU_EVCNTRn is enabled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.8 Performance Monitoring Unit Count Enable Clear Register, PMU_CNTENCLR

The PMU_CNTENCLR register disables the PMU_CCNTR register and PMU_EVCNTR0-7 registers. Reading this register shows which counters are enabled.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

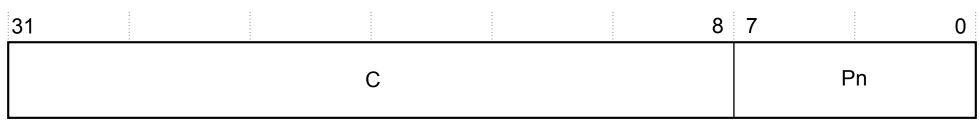
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CNTENCLR bit assignments.



The following table describes the PMU_CNTENCLR bit assignments.

Table 6-7 PMU_CNTENCLR bit assignments

Bits	Name	Function
[31:8]	C	PMU_CCNTR disable bit. The possible values are: **0** When this bit is read with this value, the cycle counter is disabled. When this bit is written to, there is no effect. **1** When this bit is read with this value, the cycle counter is enabled. When written, enables the cycle counter.
[7:0]	Pn	Event counter PMU_EVCNTR0-7 enable bit. The possible values are: `0b0` When this bit is read with this value, this implies that PMU_EVCNTRn is disabled. When this bit is written to, there is no effect. `0b1` When this bit is read with this value, this implies that PMU_EVCNTRn is enabled. When this bit is written to, PMU_EVCNTRn is disabled.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.9 Performance Monitoring Unit Interrupt Enable Set Register, PMU_INTENSET

The PMU_INTENSET register enables the generation of interrupt requests on overflows from the PMU_CCNTR, and the event counter, PMU_EVCNTR. Reading PMU_INTENSET register shows which overflow interrupt requests are enabled.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

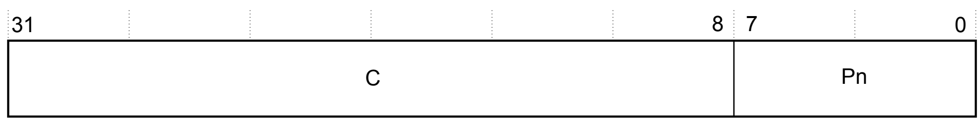
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

`0b`The following figure shows the PMU_INTENSET bit assignments.



The following table describes the PMU_INTENSET bit assignments.

Table 6-8 PMU_INTENSET bit assignments

Bits	Name	Function
[31]	C	PMU_CCNTR interrupt request enable bit. Enable the overflow interrupt for the cycle counter. The possible values of this bit are: `0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect. `0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, disables the cycle count overflow interrupt request.
[30:0]	Pn	Event counter overflow interrupt request disable bit for PMU_EVCNTRn. Disable the overflow interrupt for PMU_EVCNTRn. The possible values of this field are: `0b0` When read, means that the PMU_EVCNTRn event counter interrupt request is disabled. When written, has no effect. `0b1` When read, means that the PMU_EVCNTRn event counter interrupt request is enabled. When written, enables the PMU_EVCNTRn interrupt request. **Note** Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.10 Performance Monitoring Unit Interrupt Enable Clear Register, PMU_INTENCLR

The PMU_INTENCLR register disables the generation of interrupt requests on overflows from the PMU_CCNTR, and the event counters, PMU_EVCNTR. Reading the register shows which overflow interrupt requests are enabled.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

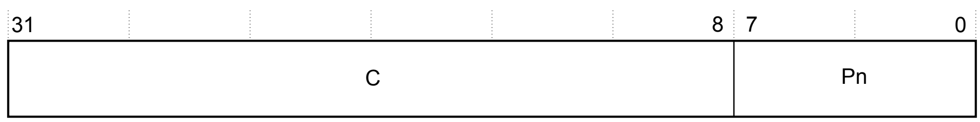
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_INTENCLR bit assignments.



The following table describes the PMU_INTENCLR bit assignments.

Table 6-9 PMU_INTENCLR bit assignments

Bits	Name	Function
[31]	C	PMU_CCNTR overflow interrupt request disable bit. Disable the overflow interrupt for the cycle counter. The possible values of this bit are: `0b0` When read, means the cycle counter overflow interrupt request is disabled. When written, has no effect. `0b1` When read, means the cycle counter overflow interrupt request is enabled. When written, enables the cycle count overflow interrupt request.
[30:0]	Pn	Event counter overflow interrupt request enable bit for PMU_EVCNTRn. Enable the overflow interrupt for PMU_EVCNTRn. The possible values of this field are: `0b0` When read, means that the PMU_EVCNTRn event counter interrupt request is disabled. When written, has no effect. `0b1` When read, means that the PMU_EVCNTRn event counter interrupt request is enabled. When written, disables the PMU_EVCNTRn interrupt request. **Note** Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.11 Performance Monitoring Unit Overflow Flag Status Clear Register, PMU_OVSCLR

The PMU_OVSCLR register contains the state of the overflow bit for the PMU_CCNTR, and each of the implemented event counters, PMU_EVCNTRn. Writing to this register clears these bits.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

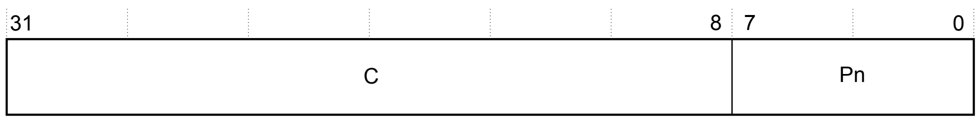
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_OVSCLR bit assignments.



The following table describes the PMU_OVSCLR bit assignments.

Table 6-10 PMU_OVSCLR bit assignments

Bits	Name	Function
[31:8]	C	PMU_CCNTR overflow bit. Clears the PMU_CCNTR overflow bit. The possible values of this bit are: `0b0` When read, means the cycle counter has not overflowed. When written, has no effect. `0b1` When read, means the cycle counter has overflowed. When written, clears the overflow bit to 0.
[7:0]	Pn	Event counter overflow clear bit for PMU_EVCNTRn. Clears the PMU_EVCNTRn overflow bit. The possible values of this field are: `0b0` When read, means that the PMU_EVCNTRn event counter has not overflowed. When written, has no effect. `0b1` When read, means that the PMU_EVCNTRn event counter has overflowed. When written, clears the PMU_EVCNTRn overflow bit to 0. **Note** Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.12 Performance Monitoring Unit Overflow Flag Status Set Register, PMU_OVSSET

The PMU_OVSSET register sets the state of the overflow bit for the PMU_CCNTR and each of the implemented event counters, PMU_EVCNTRn.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

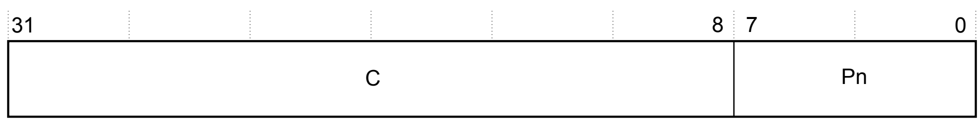
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_OVSSET bit assignments.



The following table describes the PMU_OVSSET bit assignments.

Table 6-11 PMU_OVSSET bit assignments

Bits	Name	Function
[31:8]	C	PMU_CCNTR overflow bit. Sets the overflow status for PMU_CCNTR. The possible values of this bit are: `0b0` When read, means the cycle counter has not overflowed. When written, has no effect. `0b1` When read, means the cycle counter has overflowed. When written, sets the overflow bit to 1.
[7:0]	Pn	Event counter overflow set bit for PMU_EVCNTRn. Sets the overflow status for PMU_EVCNTRn. The possible values of this field are: `0b0` When read, means that the PMU_EVCNTRn event counter has not overflowed. When written, has no effect. `0b1` When read, means that the PMU_EVCNTRn event counter has overflowed. When written, sets the PMU_EVCNTRn overflow bit to 1. **Note** Bits [30:N] are RAZ/WI, where N is the number of counters and the value of PMU_TYPE.N.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.13 Performance Monitoring Unit Software Increment Register, PMU_SWINC

The PMU_SWINC register increments a counter that is configured to count the software increment event, event `0x00`.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

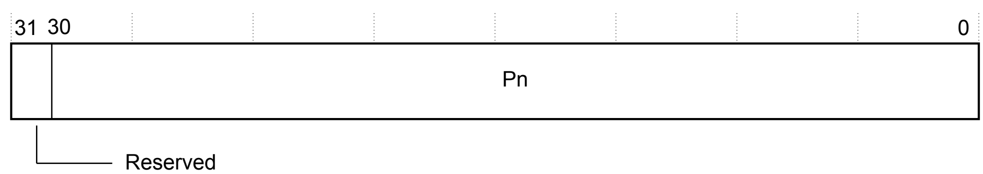
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_SWINC bit assignments.



The following table describes the PMU_SWINC bit assignments.

Table 6-12 PMU_SWINC bit assignments

Bits	Name	Function
[31]	Reserved	RES0
[30:0]	Pn	Event counter software increment bit for PMU_EVCNTRn. An event counter n, configured for SW_INCR events, increments on every write to bit n of this field. The possible values of this field are: `0b0` No action. The write to this bit is ignored. `0b1` A SW_INCR event is generated for event counter n. **Note** Bits [30:N] are WI, where N is the number of counters and the value of PMU_TYPE.N.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.14 Performance Monitoring Unit Type Register, PMU_TYPE

The PMU_TYPE register contains information regarding what the PMU supports.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

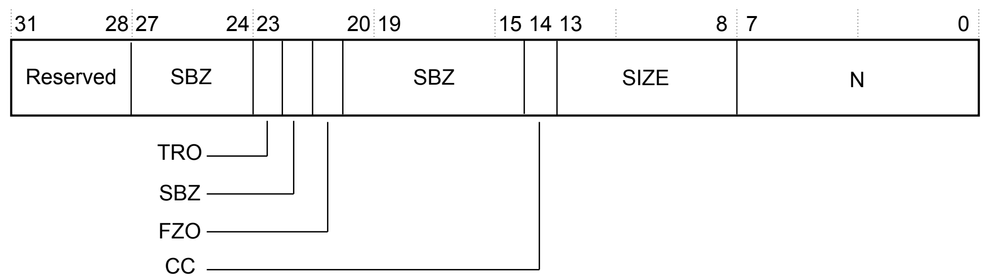
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_TYPE bit assignments.



The following table describes the PMU_TYPE bit assignments.

Table 6-13 PMU_TYPE bit assignments

Bits	Name	Function
[31:28]	Reserved	RES0
[27:24]	SBZ	SBZ
[23]	TRO	Trace-on-overflow not supported. This bit is `0b0`.
[22]	SBZ	SBZ
[21]	FZO	Freeze-on-overflow is supported. This bit is `0b1`.
[20:15]	SBZ	SBZ
[14]	CC	A dedicated cycle counter is present. This bit is `0b1`.
[13:8]	SIZE	Size of counters. This field determines the spacing of counters in the memory-map. This field is `0b011111`. **Note** This field indicates all counters are word-aligned, as the largest counter is PMU_CCNTR with 32-bits.
[7:0]	N	Number of counters. Number of counters implemented in addition to the cycle counter, PMU_CCNTR. This field is set to `0b00001000`, indicating that 8 16-bit event counters are present in addition to PMU_CCNTR.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.15 Performance Monitoring Unit Control Register, PMU_CTRL

The PMU_CTRL register configures and controls the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

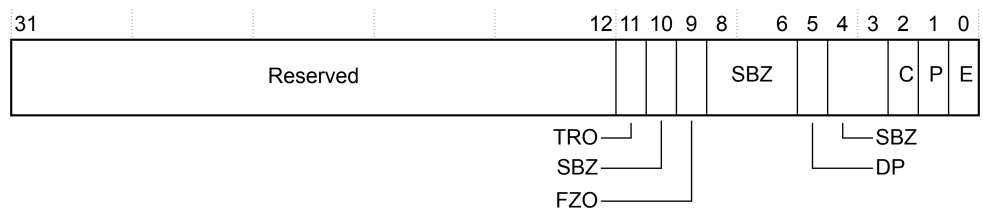
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CTRL bit assignments.



The following table describes the PMU_CTRL bit assignments.

Table 6-14 PMU_CTRL bit assignments

Bits	Name	Function
[31:12]	Reserved	RES0
[11]	TRO	Trace-on-overflow not supported in Cortex-M55. Therefore, this bit is `0b0`.
[10]	SBZ	SBZ
[9]	FZO	Freeze-on-overflow. Stops events being counted once PMU_OVSCLR or PMU_OVSSET is non-zero. The possible values of this bit are: `0b0` This bit has no effect on event counting. `0b1` While PMU_OVSCLR or PMU_OVSSET is non-zero, event counters do not count events.
[8:6]	SBZ	SBZ
[5]	DP	Disable cycle counter when event counting is prohibited.
[4:3]	SBZ	SBZ
[2]	C	Cycle counter reset. Reset the PMU_CCNTR counter. The possible values of this bit are: `0b0` No action. `0b1` Reset PMU_CCNTR to zero.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-14 PMU_CTRL bit assignments (continued)

Bits	Name	Function
[1]	P	Event counter reset. The possible values of this bit are: `0b0` No action. `0b1` Reset all event counters, except PMU_CCNTR, to zero.
[0]	E	Enable. Enable the event counters. The possible values of this bit are: `0b0` All counters, including PMU_CCNTR, are disabled. `0b1` PMU_CNTENSET enable all counters.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.16 Performance Monitoring Unit Authentication Status Register, PMU_AUTHSTATUS

The PMU_AUTHSTATUS register provides information about the state of the IMPLEMENTATION DEFINED authentication interface for the PMU

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

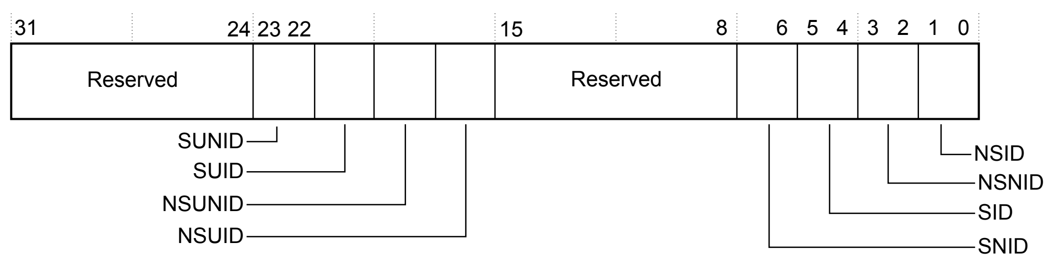
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_AUTHSTATUS bit assignments.



The following table describes the PMU_AUTHSTATUS bit assignments.

Table 6-15 PMU_AUTHSTATUS bit assignments

Bits	Name	Function
[31:24]	Reserved	RES0
[23:22]	SUNID	Secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive Debug is allowed for the Secure state. The possible values of this field are: `0b00` Unprivileged Non-invasive debug not implemented. `0b01` Reserved `0b10` Secure Non-invasive debug prohibited. `0b11` Secure Non-invasive debug allowed in unprivileged mode.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-15 PMU_AUTHSTATUS bit assignments (continued)

Bits	Name	Function
[21:20]	SUID	Secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Invasive Debug is allowed for the Secure state. The possible values of this field are: **0b00** Unprivileged Invasive debug not implemented. **0b01** Reserved **0b10** Secure Invasive debug prohibited. **0b11** Secure Invasive debug allowed in unprivileged mode.
[19:18]	NSUNID	Non-secure Unprivileged Non-invasive Debug Allowed. Indicates that Unprivileged Non-invasive Debug is allowed for the Non-secure state. The possible values of this field are: **0b00** Unprivileged Non-invasive debug not implemented. **0b01** Reserved **0b10** Non-secure Non-invasive debug prohibited. **0b11** Non-secure Non-invasive debug allowed in unprivileged mode.
[17:16]	NSUID	Non-secure Unprivileged Invasive Debug Allowed. Indicates that Unprivileged Halting Debug is allowed for the Non-secure state. The possible values of this field are: **0b00** Unprivileged halting debug not implemented. **0b01** Reserved **0b10** Non-secure halting debug prohibited. **0b11** Non-secure halting debug allowed in unprivileged mode.
[15:8]	Reserved	RES0

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table 6-15 PMU_AUTHSTATUS bit assignments (continued)

Bits	Name	Function
[7:6]	SNID	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 not allowed. `0b11` Security Extension implemented and Secure Non-invasive debug allowed.
[5:4]	SID	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 not allowed. `0b11` Security Extension implemented and Secure invasive debug allowed.
[3:2]	NSNID	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 not allowed. `0b11` Non-secure Non-invasive debug allowed.
[1:0]	NSID	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 not allowed. `0b11` Non-secure invasive debug allowed.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.17 Performance Monitoring Unit Device Architecture Register, PMU_DEVARCH

The PMU_DEVARCH register identifies the programmers model architecture of the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

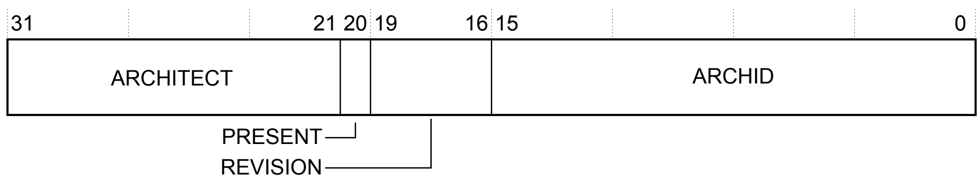
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_DEVARCH bit assignments.



The following table describes the PMU_DEVARCH bit assignments.

Table 6-16 PMU_DEVARCH bit assignments

Bits	Name	Function
[31:21]	ARCHITECT	Defines the architecture of the component. For the PMU, it is Arm Limited. Bits[31:28] are the JEP 106 continuation code, `0x4`, and bits [27:21] are the JEP 106 ID code, `0x3B`.
[20]	PRESENT	Determines the presence of DEVARCH. When set to 1, indicates that the DEVARCH is present. This bit reads as `0x1`.
[19:16]	REVISION	Defines the architecture revision. For the PMU, the revision defined by Armv8.1-M is `0x0`.
[15:0]	ARCHID	Defines this part to be an Armv8-M debug component. For the PMU, bits [15:12] are `0x0`, bits [11:0] are `0xA06`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.18 Performance Monitoring Unit Device Type Register, PMU_DEVTYPE

The PMU_DEVTYPE register indicates to a debugger that the PMU is a part of the processor interface.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

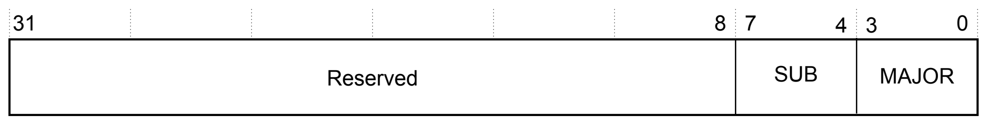
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_DEVTYPE bit assignments.



The following table describes the PMU_DEVTYPE bit assignments.

Table 6-17 PMU_DEVTYPE bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	SUB	Subtype. This field reads as `0x1`.
[3:0]	MAJOR	Major type. This field reads as `0x6`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.19 Performance Monitoring Unit Peripheral Identification Register 0, PMU_PIDR0

The PMU_PIDR0 register provides information to identify the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

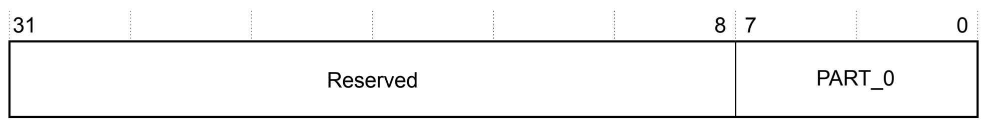
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_PIDR0 bit assignments.



The following table describes the PMU_PIDR0 bit assignments.

Table 6-18 PMU_PIDR0 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:0]	PART_0	Part number, least significant byte. This field reads `0x22`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.20 Performance Monitoring Unit Peripheral Identification Register 1, PMU_PIDR1

The PMU_PIDR1 register provides information to identify the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

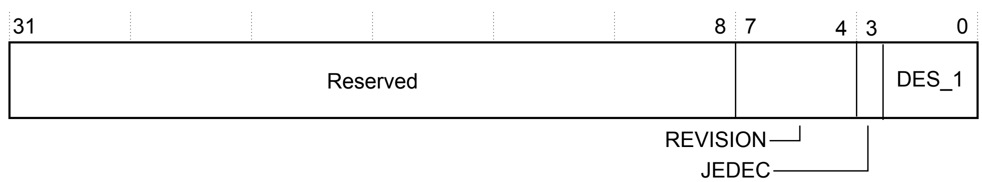
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_PIDR1 bit assignments.



The following table describes the PMU_PIDR1 bit assignments.

Table 6-19 PMU_PIDR1 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	DES_0	Designer, least significant nibble of JEP 106 ID code. For Arm Limited, this field `0xB`.
[3:0]	PART_1	Part number, most significant nibble. This field is `0xD`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.21 Performance Monitoring Unit Peripheral Identification Register 2, PMU_PIDR2

The PMU_PIDR2 register provides information to identify the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

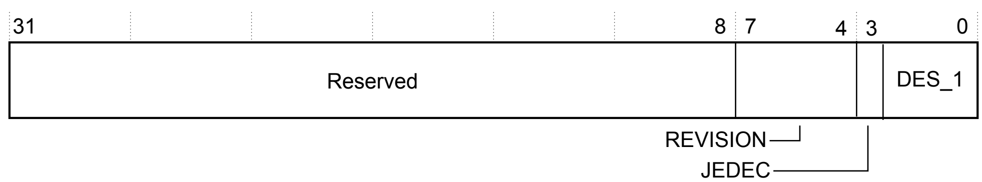
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_PIDR2 bit assignments.



The following table describes the PMU_PIDR2 bit assignments.

Table 6-20 PMU_PIDR2 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	REVISION	Part major revision. Parts can also use this field to extend Part number to 16-bits. This field reads as `0x0000`.
[3]	JEDEC	Indicates a JEP10 identity code. This is RAO.
[2:0]	DES_1	Designer, most significant bits of JEP 106 ID code. For Arm Limited, this field `0b011`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.22 Performance Monitoring Unit Peripheral Identification Register 3, PMU_PIDR3

The PMU_PIDR3 register provides information to identify the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

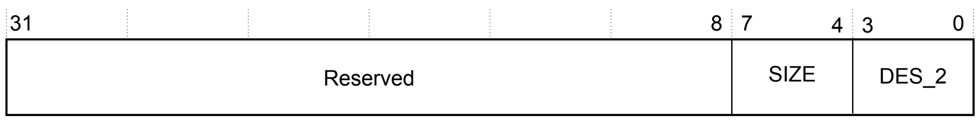
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_PIDR3 bit assignments.



The following table describes the PMU_PIDR3 bit assignments.

Table 6-21 PMU_PIDR3 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	REVAND	Part minor revision. This field is `0x0000`.
[3:0]	CMOD	Customer modified. This field is `0x0000`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.23 Performance Monitoring Unit Peripheral Identification Register 4, PMU_PIDR4

The PMU_PIDR4 register provides information to identify the PMU.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

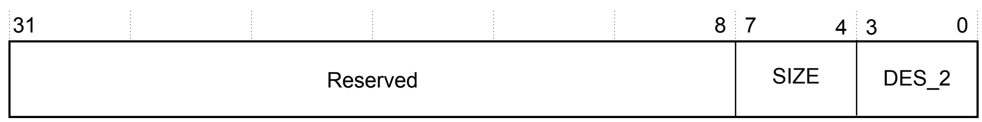
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_PIDR4 bit assignments.



The following table describes the PMU_PIDR4 bit assignments.

Table 6-22 PMU_PIDR4 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	SIZE	Size of the component. This field is RAZ.
[3:0]	DES_2	Designer, JEP 106 continuation code, least significant nibble. For Arm Limited, this field is `0x4`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.24 Performance Monitoring Unit Component Identification Register 0, PMU_CIDR0

The PMU_CIDR0 register provides information to identify a PMU component.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

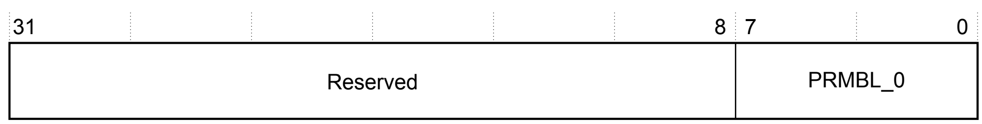
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CIDR0 bit assignments.



The following table describes the PMU_CIDR0 bit assignments.

Table 6-23 PMU_CIDR0 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:0]	PRMBL_0	Preamble. This field reads `0x0D`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.25 Performance Monitoring Unit Component Identification Register 1, PMU_CIDR1

The PMU_CIDR1 register provides information to identify a PMU component.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

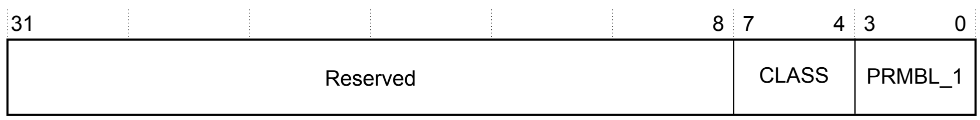
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CIDR1 bit assignments.



The following table describes the PMU_CIDR1 bit assignments.

Table 6-24 PMU_CIDR1 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[7:4]	CLASS	Component class. This field reads `0x0`.
[3:0]	PRMBL_1	Preamble. This field reads `0x0`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.26 Performance Monitoring Unit Component Identification Register 2, PMU_CIDR2

The PMU_CIDR2 register provides information to identify a PMU component.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

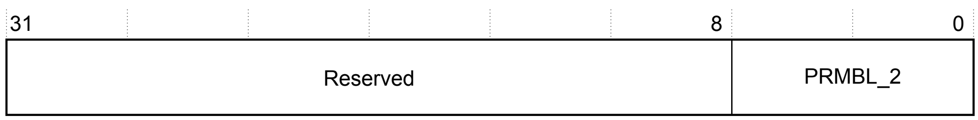
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CIDR2 bit assignments.



The following table describes the PMU_CIDR2 bit assignments.

Table 6-25 PMU_CIDR2 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[3:0]	PRMBL_2	Preamble. This field reads `0x05`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

6.27 Performance Monitoring Unit Component Identification Register 3, PMU_CIDR3

The PMU_CIDR3 register provides information to identify a PMU component.

Usage Constraints

Privileged access only. See *6.3 PMU register summary* on page 6-660 for more information.

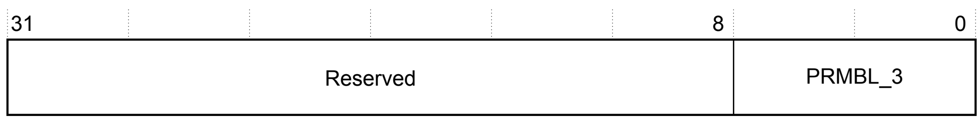
Configurations

Present only if the PMU is implemented.

Attributes

If the Security Extension is implemented, this register is not banked between security states. See *4.7 Implementation defined register summary* on page 4-575 for more information.

The following figure shows the PMU_CIDR3 bit assignments.



The following table describes the PMU_CIDR3 bit assignments.

Table 6-26 PMU_CIDR3 bit assignments

Bits	Name	Function
[31:8]	Reserved	RES0
[3:0]	PRMBL_3	Preamble. This field reads `0xB1`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix A
External Wakeup Interrupt Controller

This appendix describes the *External Wakeup Interrupt Controller* (EWIC) that can be optionally implemented with the processor.

It contains the following sections:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.1 EWIC features

The processor supports the *External Wakeup Interrupt Controller* (EWIC), which is a peripheral to the system and is suitable for sleep states where the entire processor sub-system is powered down. The EWIC does not control powerdown, however, it allows enough state to be saved for wake-up from a powered down state.

An APB interface controls the EWIC which must be connected to the *External Private Peripheral Bus* (EPPB) master interface of the processor. This interface is used to communicate all interrupt and event status information on sleep entry and wakeup. The EWIC interface can be asynchronous to the processor by instantiating synchronizers in the system on the APB interface.

EWIC configuration

The EWIC can be configured to support a variable number of events.

A minimum of 4 events are supported:

- External event.
- Debug request.
- *Non-Maskable Interrupt* (NMI).
- One interrupt.

A maximum of 483 events are supported:

- External event.
- Debug request.
- NMI.
- 480 interrupts.

Any number of events in the range 4-483 is permitted.

Note

The EWIC can support fewer interrupts than the processor supports. Interrupts above those that the EWIC supports cannot cause the core to exit low-power state. Therefore, higher numbered interrupts that occur when the core is in a low-power state might be lost.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.2 EWIC register summary

The *External Wakeup Interrupt Controller* (EWIC) requires memory-mapped registers that are accessed at address `0xE0047000` onwards in the PPB region of the memory map. The registers are contained in a CoreSight compliant 4KB block. The following table shows the EWIC registers.

Table A-1 EWIC register summary

Address	Name	Type	Reset value	Description
`0xE0047000`	EWIC_CR	RW	`0x00000000`	*A.3 EWIC Control Register* on page Appx-A-693
`0xE0047004`	EWIC_ASCR	RW	`0x00000003`	*A.4 EWIC Automatic Sequence Control Register* on page Appx-A-694
`0xE0047008`	EWIC_CLRMASK	WO	`0x00000000`	*A.5 EWIC Clear Mask Register* on page Appx-A-696
`0xE004700C`	EWIC_NUMID	RO	UNKNOWN	*A.6 EWIC Event Number ID Register* on page Appx-A-697
`0xE0047200`	EWIC_MASKA	RW	UNKNOWN	*A.7 EWIC Mask Registers* on page Appx-A-698
`0xE0047204` - `0xE004723C`	EWIC_MASKn	RW	UNKNOWN	
`0xE0047400`	EWIC_PENDA	RO	UNKNOWN	*A.8 EWIC Pend Event Registers* on page Appx-A-700
`0xE0047404` - `0xE004743C`	EWIC_PENDn	RW	UNKNOWN	
`0xE0047600`	EWIC_PSR	RO	UNKNOWN	*A.9 EWIC Pend Summary Register* on page Appx-A-702
`0xE0047604-0xE0047EFC`	-	UNK/SBZP	-	Reserved
`0xE0047F00-0xE0047FFC`	CoreSight registers	RO		*A.10 EWIC CoreSight™ register summary* on page Appx-A-703

Note

The processor controls access to the EWIC peripheral registers.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.3 EWIC Control Register

The EWIC_CR is the main *External Wakeup Interrupt Controller* (EWIC) control register.

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, the Cortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_CR bit assignments.

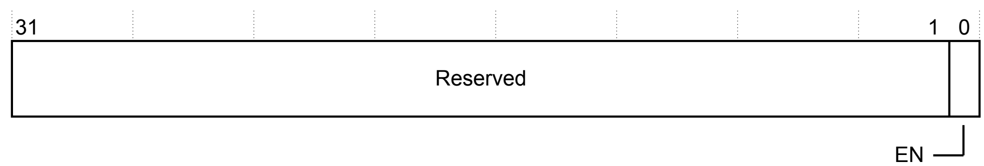


Figure A-1 EWIC_CR bit assignments

The following table describes the EWIC_CR bit assignments.

Table A-2 EWIC_CR bit assignments

Field	Name	Type	Description
[31:1]	-	-	Reserved, RES0
[0]	EN	RW	The options are: **0** EWIC is disabled, events are not pended, and **WAKEUP** is not signaled. **1** EWIC is enabled, events are pended, and **WAKEUP** is signaled. The reset value is 0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.4 EWIC Automatic Sequence Control Register

The EWIC_ASCR determines whether the processor generates APB transactions on entry and exit from *Wakeup Interrupt Controller* (WIC) sleep to set up the wakeup state in the *External Wakeup Interrupt Controller* (EWIC).

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, the Cortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_ASCR bit assignments.

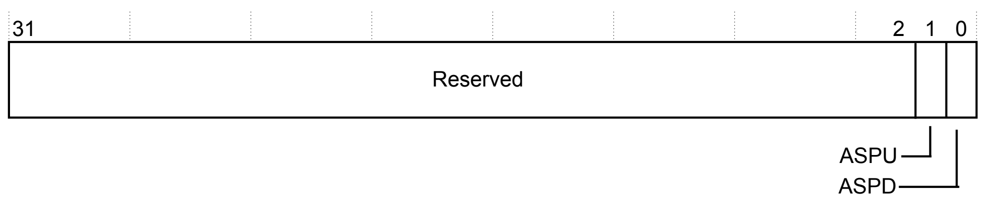


Figure A-2 EWIC_ASCR bit assignments

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

The following table describes the EWIC_ASCR bit assignments.

Table A-3 EWIC_ASCR bit assignments

Field	Name	Type	Description
[31:2]	-	-	Reserved, RES0
[1]	ASPU	RW	The value of this bit is sent to the processor. The processor must use this value to decide whether any automatic EWIC accesses must be performed on transitioning from a low-power state. The options are: **0** No automatic sequence on power up. **1** Automatic sequence on powerup. The reset value is 1.
[0]	ASPD	RW	The value of this bit is sent to the processor. The processor must use this value to decide whether any automatic EWIC accesses must be performed on transitioning to a low-power state. The options are: **0** No automatic sequence on entry to a low-power state. **1** Automatic sequence on entry to a low-power state. The reset value is 1.

Note

- If the automatic sequence is disabled, then software can program the unit by writing to the EWIC_MASKA and EWIC_MASKn registers on sleep entry and reading from the EWIC_PENDn registers on sleep exit.
- The value of EWIC_ASCR does not affect the operation of the EWIC itself. It only affects the control information that is driven on the **WICCONTROL** signal to the Cortex-M55 processor.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.5 EWIC Clear Mask Register

When there are writes to the EWIC_CLRMASK register, it causes EWIC_MASKA and all the EWIC_MASKn registers to be cleared. The write data is ignored. This register is RAZ.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.6 EWIC Event Number ID Register

The EWIC_NUMID register returns the number of events supported in the *External Wakeup Interrupt Controller* (EWIC) that have been configured during synthesis.

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, the Cortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_NUMID bit assignments.

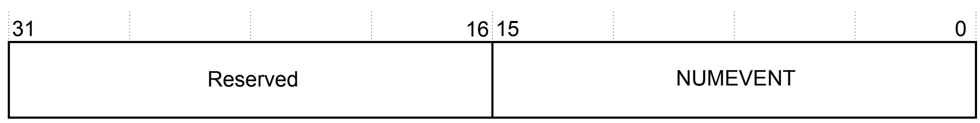


Figure A-3 EWIC_NUMID bit assignments

The following table describes the EWIC_NUMID bit assignments.

Table A-4 EWIC_NUMID bit assignments

Field	Name	Type	Description
[31:16]	-	-	Reserved, RES0
[15:0]	NUMEVENT	RO	The number of events supported.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.7 EWIC Mask Registers

The EWIC_MASKA register defines the mask for special events and the EWIC_MASKn registers for external interrupt (IRQ) events. There is one EWIC_MASKn register implemented for every 32 events that the *External Wakeup Interrupt Controller* (EWIC) supports. At least one register is always implemented. EWIC_MASKn is at address `0xE0047204`+(n×4).

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, the Cortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

These registers is always implemented when the EWIC is included.

Attributes

These are 32-bit registers.

The following figure shows the EWIC_MASKA bit assignments.

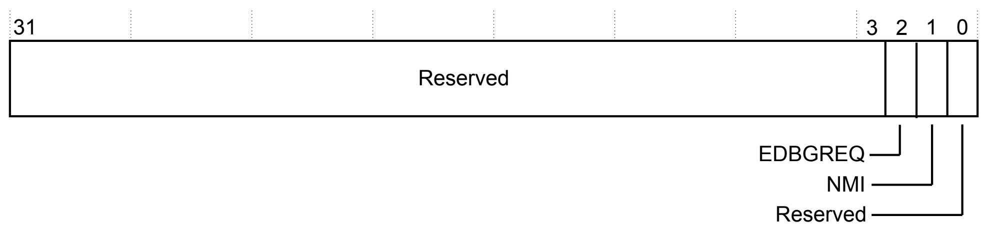


Figure A-4 EWIC_MASKA bit assignments

The following table describes the EWIC_MASKA bit assignments.

Table A-5 EWIC_MASKA bit assignments

Field	Name	Type	Description
[31:3]	-	-	Reserved, RES0
[2]	EDBGREQ	RW	Mask for external debug request.
[1]	NMI	RW	Mask for *Non-Maskable Interrupt* (NMI).
[0]	EVENT	RW	Mask for *Wait For Exception* (WFE) wakeup event.

The following figure shows the EWIC_MASKn, where n=0-14, bit assignments.

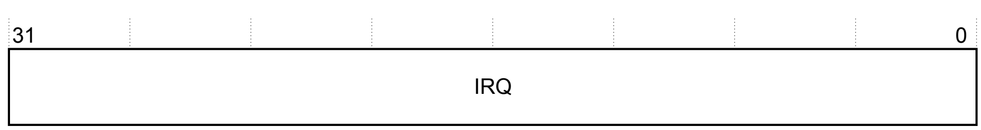


Figure A-5 EWIC_MASKn, where n=0-14 bit assignments

The following table describes the EWIC_MASKn, where n=0-14, bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table A-6 EWIC_MASKn, where n=0-14, bit assignments

Field	Name	Type	Description
[31:0]	IRQ	RW	Masks for interrupts (n×32) to ((n+1)×32)-1

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.8 EWIC Pend Event Registers

These registers indicate which events have been pended. The EWIC_PENDA register is used for special events and the EWIC_PENDn registers are used for external interrupt (IRQ) events. There is one EWIC_PENDn registers implemented for each 32 interrupt events the EWIC supports. EWIC_PENDA and at least one EWIC_PENDn register is always implemented.

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, the Cortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

These registers are always implemented when the EWIC is included. There is one EWIC_PENDn register implemented for every 32 events that the *External Wakeup Interrupt Controller* (EWIC) supports. At least one register is always implemented. EWIC_MASKn is at address `0xE0047404`+(n×4).

Attributes

These are 32-bit registers. The EWIC_PENDn registers can be written to transfer pended interrupts in the NVIC when the processor enters sleep. EWIC_PENDA is read-only as special events can only be pended by the system (usually during sleep).

The following figure shows the EWIC_PENDA bit assignments.

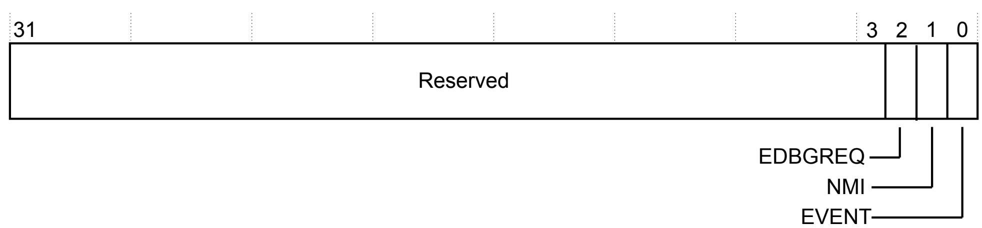


Figure A-6 EWIC_PENDA bit assignments

The following table describes the EWIC_PENDA bit assignments.

Table A-7 EWIC_PENDA bit assignments

Field	Name	Type	Description
[31:3]	-	-	Reserved, RES0
[2]	EDBGREQ	RO	External debug request is pended.
[1]	NMI	RO	*Non-Maskable Interrupt* (NMI) is pended.
[0]	EVENT	RO	*Wait For Exception* (WFE) wakeup event is pended.

The following figure shows the EWIC_PENDn, where n=0-14, bit assignments.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

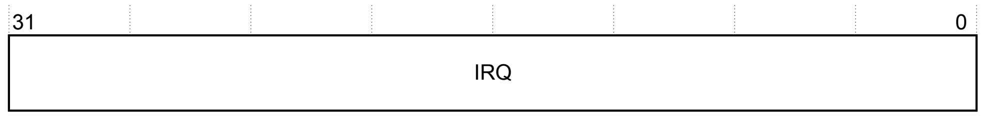


Figure A-7 EWIC_PENDn, where n=0-14 bit assignments

The following table describes the EWIC_PENDn, where n=0-14, bit assignments.

Table A-8 EWIC_PENDn, where n=0-14, bit assignments

Field	Name	Type	Description
[31:0]	IRQ	RW	Interrupts (n×32) to ((n+1)×32)-1 are pended. A write of zero to this field is ignored.

Note

Any IRQ bit associated with an interrupt that the EWIC does not support is RAZ/WI. All EWIC_PENDn registers are reset 0. If an event occurs when EWIC_CR.EN is set, then the corresponding identical bit in EWIC_PENDn is set. All EWIC_PENDn registers are cleared if the EWIC is disabled, that is, if EWIC_CR.EN is clear.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.9 EWIC Pend Summary Register

The EWIC_PSR indicates which EWIC_PENDn registers are non-zero. This allows the processor to efficiently determine which EWIC_PENDn registers need to be read. This can be used to improve code efficiency in the power on sequence.

Usage constraints

When the EWIC is connected to the *External Private Peripheral Bus* (EPPB) interface, theCortex-M55 processor controls access to these registers using the following constraints:

- If the Armv8.1-M Security Extension is included, then access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access from Non-secure software is only allowed if AIRCR.BFHFNMINS is set to 1.
- Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PSR bit assignments.

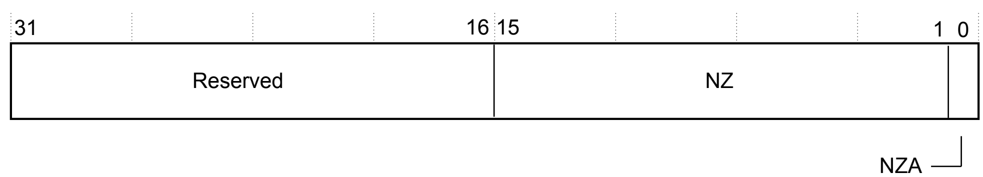


Figure A-8 EWIC_PSR bit assignments

The following table describes the EWIC_PSR bit assignments.

Table A-9 EWIC_PSR bit assignments

Field	Name	Type	Description
[31:16]	-	-	Reserved, RES0
[15:1]	NZ	RO	If EWIC_PSR.NZ[n+1] is set, then EWIC_PENDn is non-zero.
[0]	NZA	RO	If EWIC_PSR.NZA set, then EWIC_PENDA is non-zero

Note

If any bit of EWIC_PSR is associated with an EWIC_PENDn register that is entirely RAZ/WI, then the bit in EWIC_PSR is also RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.10 EWIC CoreSight™ register summary

The *External Wakeup Interrupt Controller* (EWIC) implements the standard CoreSight registers.

The following table describes the CoreSight registers that the EWIC implements.

Table A-10 EWIC CoreSight register summary

Address	Name	Type	Reset value	Description
0xE0047F00	EWIC_ITCTRL	RO	0x00000000	*A.11 EWIC Integration Mode Control Register* on page Appx-A-705
0xE0047F04-0xE0047F9C	-	-	-	Reserved
0xE0047FA0	EWIC_CLAIMSET	RW	0x0000000F	*A.12 EWIC Claim Tag Set Register* on page Appx-A-706
0xE0047FA4	EWIC_CLAIMCLR	RW	0x00000000	*A.13 EWIC Claim Tag Clear Register* on page Appx-A-707
0xE0047FA8	EWIC_DEVAFF0	RO	0x80000000	*A.14 EWIC Device Affinity Register 0* on page Appx-A-708
0xE0047FAC	EWIC_DEVAFF1	RO	0x00000000	*A.15 EWIC Device Affinity Register 1* on page Appx-A-709
0xE0047FB0	EWIC_LAR	WO	UNKNOWN	*A.16 EWIC Software Lock Access Register* on page Appx-A-710
0xE0047FB4	EWIC_LSR	RO	0x00000000	*A.17 EWIC Software Lock Status Register* on page Appx-A-711
0xE0047FB8	EWIC_AUTHSTATUS	RO	0x00000000	*A.18 EWIC Authentication Status Register* on page Appx-A-712
0xE0047FBC	EWIC_DEVARCH	RO	0x47700A07	*A.19 EWIC Device Architecture Register* on page Appx-A-713
0xE0047FC0	EWIC_DEVID2	RO	0x00000000	*A.20 EWIC Device Configuration Register 2* on page Appx-A-714
0xE0047FC4	EWIC_DEVID1	RO	0x00000000	*A.21 EWIC Device Configuration Register 1* on page Appx-A-715
0xE0047FC8	EWIC_DEVID	RO	0x00000000	*A.22 EWIC Device Configuration Register* on page Appx-A-716
0xE0047FCC	EWIC_DEVTYPE	RO	0x00000000	*A.23 EWIC Device Type Identifier Register, EWIC_DEVTYPE* on page Appx-A-717
0xE0047FD0	EWIC_PIDR4	RO	0x00000004	*A.24 Peripheral Identification Register 4, EWIC_PIDR4* on page Appx-A-718
0xE0047FD4	EWIC_PIDR5	RO	0x00000000	*A.25 Peripheral Identification Register 5, EWIC_PIDR5* on page Appx-A-719
0xE0047FD8	EWIC_PIDR6	RO	0x00000000	*A.26 Peripheral Identification Register 6, EWIC_PIDR6* on page Appx-A-720
0xE0047FDC	EWIC_PIDR7	RO	0x00000000	*A.27 Peripheral Identification Register 7, EWIC_PIDR7* on page Appx-A-721
0xE0047FE0	EWIC_PIDR0	RO	0x00000022	*A.28 Peripheral Identification Register 0, EWIC_PIDR0* on page Appx-A-722

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Table A-10 EWIC CoreSight register summary (continued)

Address	Name	Type	Reset value	Description
0xE0047FE4	EWIC_PIDR1	RO	0x000000BD	*A.29 Peripheral Identification Register 1, EWIC_PIDR1* on page Appx-A-723
0xE0047FE8	EWIC_PIDR2	RO	0x0000000B	*A.30 Peripheral Identification Register 2, EWIC_PIDR2* on page Appx-A-724
0xE0047FEC	EWIC_PIDR3	RO	0x00000000	*A.31 Peripheral Identification Register 3, EWIC_PIDR3* on page Appx-A-725
0xE0047FF0	EWIC_CIDR0	RO	0x0000000D	*A.32 Component Identification Register 0, EWIC_CIDR0* on page Appx-A-726
0xE0047FF4	EWIC_CIDR1	RO	0x00000090	*A.33 Component Identification Register 1, EWIC_CIDR1* on page Appx-A-727
0xE0047FF8	EWIC_CIDR2	RO	0x00000005	*A.34 Component Identification Register 2, EWIC_CIDR2* on page Appx-A-728
0xE0047FFC	EWIC_CIDR3	RO	0x000000B1	*A.35 Component Identification Register 3, EWIC_CIDR3* on page Appx-A-729

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.11 EWIC Integration Mode Control Register

The EWIC_ITCTRL register is used to dynamically switch between functional mode and integration mode. In integration mode, topology detection is enabled. The EWIC does not support integration mode, and this register is RAZ.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.12 EWIC Claim Tag Set Register

The EWIC_CLAIMSET register is used to set whether functionality is in use by a debug agent. The EWIC does not have any associated debug functionality, and this register is `0xF`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.13 EWIC Claim Tag Clear Register

The EWIC_CLAIMCLR register is used to set whether functionality is in use by a debug agent. The EWIC does not have any associated debug functionality, and this register is `0x0`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.14 EWIC Device Affinity Register 0

The EWIC_DEVAFF0 register enables a debugger to determine whether the EWIC and the processor have an affinity with each other. The EWIC does not have any associated debug functionality, and this register is `0x80000000`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.15 EWIC Device Affinity Register 1

The EWIC_DEVAFF1 register enables a debugger to determine whether the EWIC and the processor have an affinity with each other. The EWIC does not have any associated debug functionality, and this register is `0x0`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.16 EWIC Software Lock Access Register

The EWIC_LAR register controls software access to CoreSight components to reduce the likelihood of accidental access to the EWIC. The EWIC does not support software locking, and writing to this register has no affect.

For more information on the implications of software access and locking, see the *Arm® CoreSight™ Architecture Specification v3.0*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.17 EWIC Software Lock Status Register

The EWIC_LSR register controls software access to CoreSight components to reduce the likelihood of accidental access to the EWIC. The EWIC does not support software locking, and this register is RAZ.

For more information on the implications of software access and locking, see the *Arm® CoreSight™ Architecture Specification v3.0*.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.18 EWIC Authentication Status Register

The EWIC_AUTHSTATUS register indicates whether certain debug functions are enabled for the EWIC component. The EWIC is not a debug component, therefore, this register is RAZ.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.19 EWIC Device Architecture Register

The EWIC_DEVARCH identifies the architecture and architecture of the EWIC.

For more information on the implications of software access and locking, see the *Arm® CoreSight™ Architecture Specification v3.0*.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_DEVARCH bit assignments.

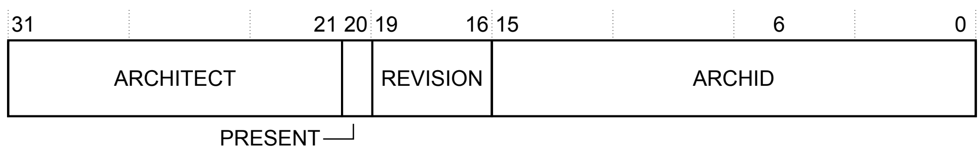


Figure A-9 EWIC_DEVARCH bit assignments

The following table describes the EWIC_DEVARCH bit assignments.

Table A-11 EWIC_DEVARCH bit assignments

Field	Name	Type	Description
[31:21]	ARCHITECT	RO	Defines the architect of the component: **Bits[31:28]** Indicates the JEP106 continuation code. **Bits[27:21]** Indicates the JEP106 identification code. The value of this field is `0b01000111011`.
[20]	PRESENT	RO	Indicates the presence of this register. This value is `0b1`.
[19:16]	REVISION	RO	Architecture revision. This value is `0b0000`
[15:0]	ARCHID	RO	Architecture ID. This value is `0x0A07`

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.20 EWIC Device Configuration Register 2

The EWIC_DEVID2 indicates capabilities of the EWIC. This register is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.21 EWIC Device Configuration Register 1

The EWIC_DEVID1 indicates capabilities of the EWIC. This register is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.22 EWIC Device Configuration Register

The EWIC_DEVID indicates capabilities of the EWIC. This register is RAZ/WI.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.23 EWIC Device Type Identifier Register, EWIC_DEVTYPE

The EWIC_DEVTYPE register provides part number information about the EWIC component to the debugger.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_AUTHSTATUS bit assignments.

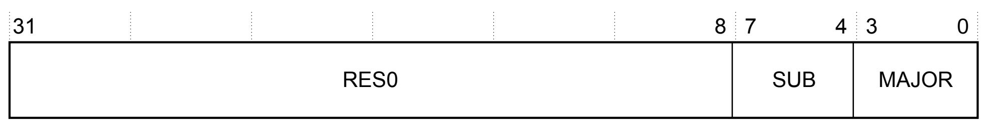


Figure A-10 EWIC_AUTHSTATUS bit assignments

The following table describes the EWIC_AUTHSTATUS bit assignments.

Table A-12 EWIC_AUTHSTATUS bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0
[7:4]	SUB	RO	Sub type for the component device type. This field is set to `0b0000`.
[3:0]	MAJOR	RO	Major type for the component device type. This field is set to `0b0000`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.24 Peripheral Identification Register 4, EWIC_PIDR4

The EWIC_PIDR4 register provides information about the memory size and JEP106 continuation code that the *External Wakeup Interrupt Controller* (EWIC) component uses.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR4 bit assignments.

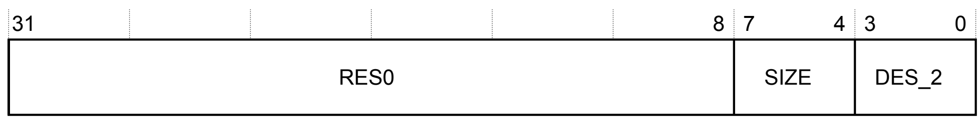


Figure A-11 EWIC_PIDR4 bit assignments

The following table describes the EWIC_PIDR4 bit assignments.

Table A-13 EWIC_PIDR4 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:4]	SIZE	RO	This field indicates the memory size that the EWIC uses. This field returns `0x0` indicating that the component uses an UNKNOWN number of 4KB blocks. The reset value of this field is `0x0`.
[3:0]	DES_2	RO	JEP106 continuation code. Together with EWIC_PIDR2.DES_1 and EWIC_PIDR1.DES_0, they indicate the designer of the component, not the implementer, except where the two are the same. The reset value of this field is `0x4`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.25 Peripheral Identification Register 5, EWIC_PIDR5

The EWIC_PIDR5 register is reserved.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR5 bit assignments.

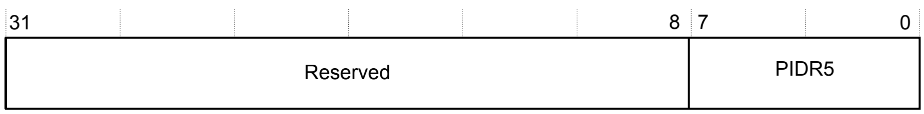


Figure A-12 EWIC_PIDR5 bit assignments

The following table describes the EWIC_PIDR5 bit assignments.

Table A-14 EWIC_PIDR5 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PIDR5	RO	RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.26 Peripheral Identification Register 6, EWIC_PIDR6

The EWIC_PIDR6 register is reserved.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR6 bit assignments.

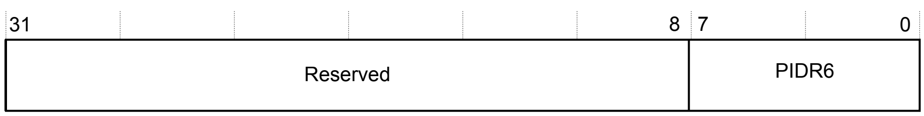


Figure A-13 EWIC_PIDR6 bit assignments

The following table describes the EWIC_PIDR6 bit assignments.

Table A-15 EWIC_PIDR6 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PIDR6	RO	RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.27 Peripheral Identification Register 7, EWIC_PIDR7

The EWIC_PIDR7 register is reserved.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR7 bit assignments.

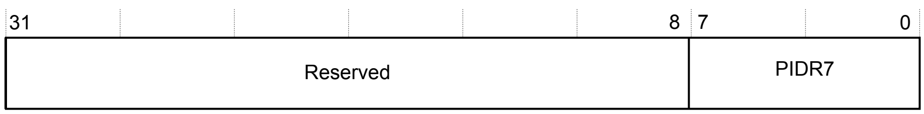


Figure A-14 EWIC_PIDR7 bit assignments

The following table describes the EWIC_PIDR7 bit assignments.

Table A-16 EWIC_PIDR7 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PIDR7	RO	RES0.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.28 Peripheral Identification Register 0, EWIC_PIDR0

The EWIC_PIDR0 register indicates the EWIC component part number.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR0 bit assignments.

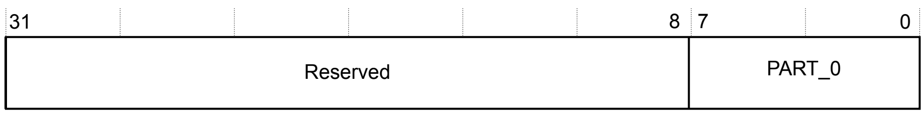


Figure A-15 EWIC_PIDR0 bit assignments

The following table describes the EWIC_PIDR0 bit assignments.

Table A-17 EWIC_PIDR0 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PART_0	RO	This field indicates the part number. When taken together with EWIC_PIDR1.PART_1, it indicates the component. The part number is selected by the designer of the component. The reset value of this field is `0b00100010`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.29 Peripheral Identification Register 1, EWIC_PIDR1

The EWIC_PIDR1 register indicates the EWIC component JEP106 continuation code and part number.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR1 bit assignments.

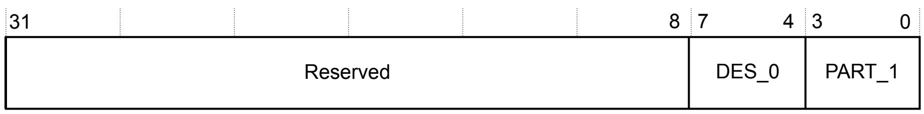


Figure A-16 EWIC_PIDR1 bit assignments

The following table describes the EWIC_PIDR1 bit assignments.

Table A-18 EWIC_PIDR1 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:4]	DES_0	RO	This field indicates the JEP106 identification code, bits[3:0]. Together, with EWIC_PIDR4.DES_2 and EWIC_PIDR2.DES_1, they indicate the designer of the component and not the implementer, except where the two are the same. The reset value is `0xB`.
[3:0]	PART_1	RO	This field indicates the part number, bits[11:8]. Taken together with EWIC_PIDR0.PART_0 it indicates the component. The part number is selected by the designer of the component. The reset value is `0xD`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.30 Peripheral Identification Register 2, EWIC_PIDR2

The EWIC_PIDR2 register indicates the EWIC component revision number, JEDEC value, and part of the JEP106 continuation code.

Usage constraints

Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_PIDR2 bit assignments.

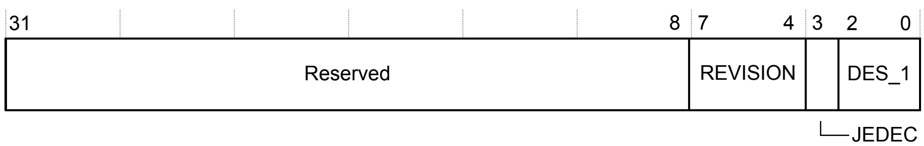


Figure A-17 EWIC_PIDR2 bit assignments

The following table describes the EWIC_PIDR2 bit assignments.

Table A-19 EWIC_PIDR2 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:4]	REVISION	RO	This field indicates the revision number of the EWIC component. It is an incremental value starting at `0x0` for the first design. The reset value is `0x0`.
[3]	JEDEC	RO	This field is always 1, indicating that a JEDEC assigned value is used.
[2:0]	DES_1	RO	This field is the JEP106 identification code, bits[6:4]. Together, with CTI_PIDR4.DES_2 and CTI_PIDR1.DES_0, they indicate the designer of the component and not the implementer, except where the two are the same. The reset value is `0b011`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.31 Peripheral Identification Register 3, EWIC_PIDR3

The CTI_PIDR3 register indicates minor errata fixes of the *Cross Trigger Interface* (CTI) component and if you have modified the behavior of the component.

Usage constraints

Access is only allowed from privileged code. Unprivileged access results in a BusFault being raised.

Configurations

This register is always implemented when the CTI is included.

Attributes

This is a 32-bit register.

The following figure shows the CTI_PIDR3 bit assignments.

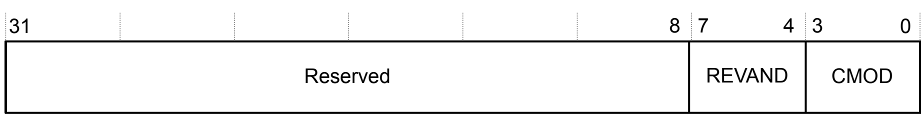


Figure A-18 CTI_PIDR3 bit assignments

The following table describes the CTI_PIDR3 bit assignments.

Table A-20 CTI_PIDR3 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:4]	REVAND	RO	This field indicates minor errata fixes specific to this design, for example metal fixes after implementation. In most cases this field is `0x0`.
[3:0]	CMOD	RO	Customer modified. Where the component is reusable IP, this value indicates whether you have modified the behavior of the component. In most cases, this field is `0x0`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.32 Component Identification Register 0, EWIC_ CIDR0

The EWIC_CIDR0 register indicates the preamble.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_CIDR0 bit assignments.

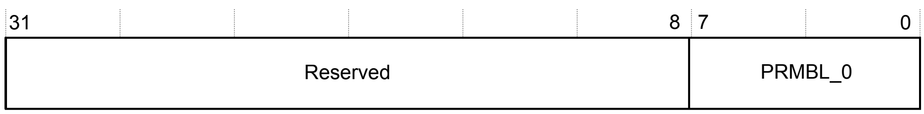


Figure A-19 EWIC_CIDR0 bit assignments

The following table describes the EWIC_CIDR0 bit assignments.

Table A-21 EWIC_CIDR0 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PRMBL_0	RO	Preamble. This field returns `0x0D`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.33 Component Identification Register 1, EWIC_ CIDR1

The EWIC_CIDR1 register indicates the component class and preamble.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_CIDR1 bit assignments.

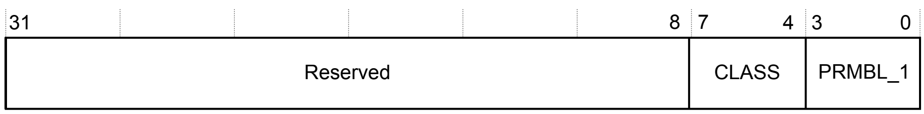


Figure A-20 EWIC_CIDR1 bit assignments

The following table describes the EWIC_CIDR1 bit assignments.

Table A-22 EWIC_CIDR1 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:4]	CLASS	RO	Component class. Returns `0x9`, indicating this is a CoreSight component.
[3:0]	PRMBL_1	RO	Preamble. This field returns `0x0`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.34 Component Identification Register 2, EWIC_ CIDR2

The EWIC_CIDR2 register indicates the preamble.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_CIDR2 bit assignments.

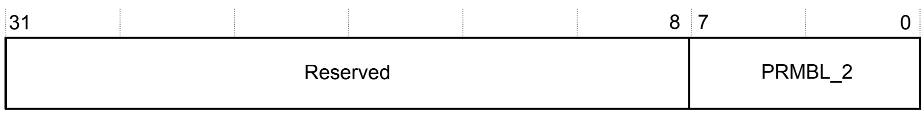


Figure A-21 EWIC_CIDR2 bit assignments

The following table describes the EWIC_CIDR2 bit assignments.

Table A-23 EWIC_CIDR2 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PRMBL_2	RO	Preamble. This field returns `0x05`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

A.35 Component Identification Register 3, EWIC_ CIDR3

The EWIC_CIDR3 register indicates the preamble.

Usage constraints

See *A.10 EWIC CoreSight™ register summary* on page Appx-A-703 and *Arm® CoreSight™ Architecture Specification v3.0* for more information.

Configurations

This register is always implemented when the EWIC is included.

Attributes

This is a 32-bit register.

The following figure shows the EWIC_CIDR3 bit assignments.

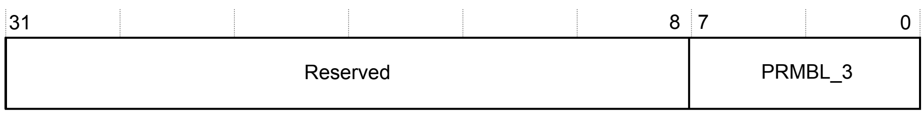


Figure A-22 EWIC_CIDR3 bit assignments

The following table describes the EWIC_CIDR3 bit assignments.

Table A-24 EWIC_CIDR3 bit assignments

Field	Name	Type	Description
[31:8]	Reserved	-	RES0.
[7:0]	PRMBL_3	RO	Preamble. This field returns `0xB1`.

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

Appendix B
Revisions

This appendix describes the technical changes between released issues of this book.

It contains the following section:

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential

B.1 Revisions

The following tables show any significant technical changes between released issues of this book.

Table B-1 Issue 0001-01

Change	Location	Updated in version
First early access release for r0p1	-	-

Copyright © 2020 Arm Limited or its affiliates. All rights reserved.
Non-Confidential