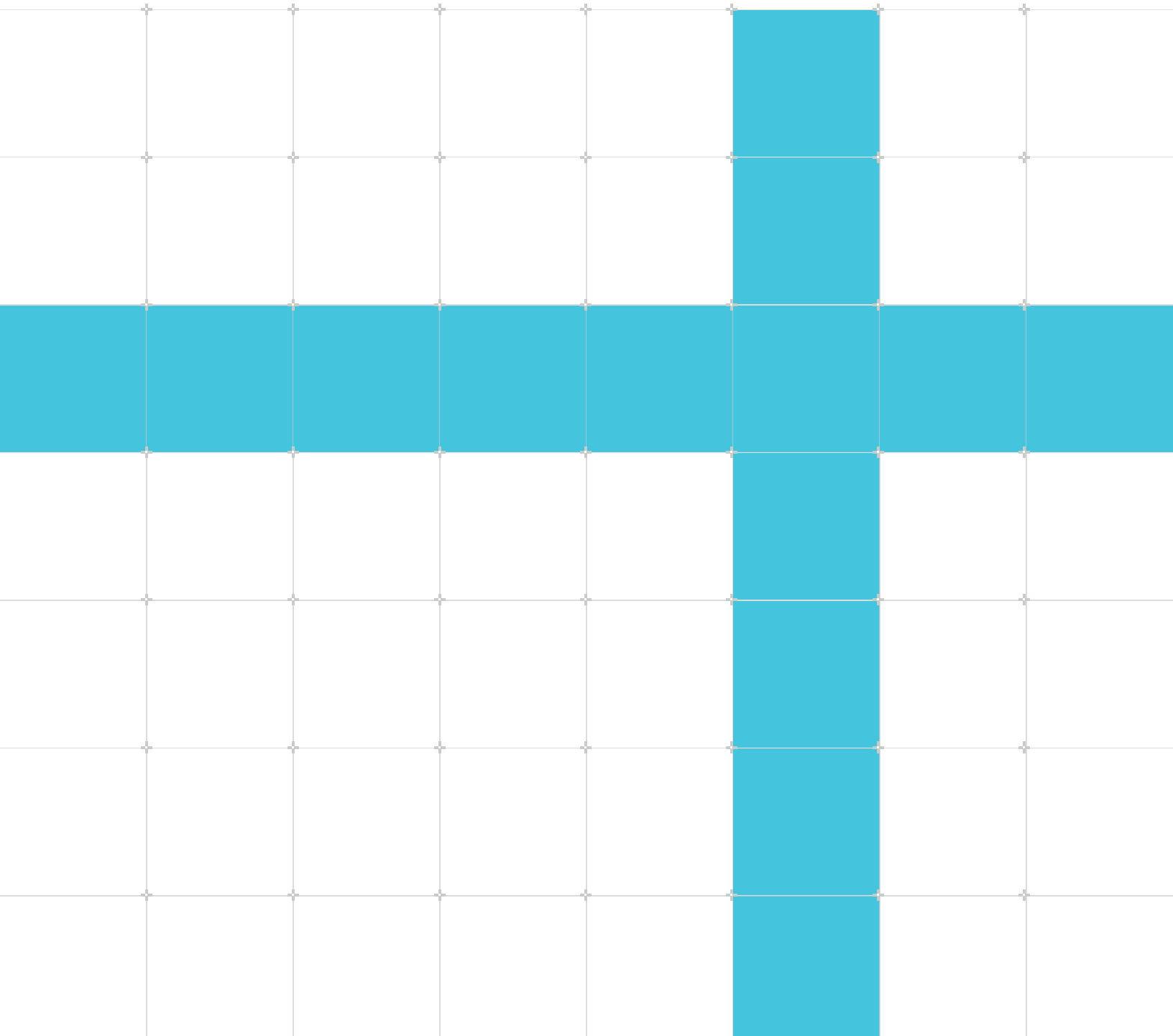
Arm® Cortex®-M33 Devices

Revision: r1p0

Generic User Guide

Issue 05
100235_0100_05_en



Arm® Cortex®-M33 Devices

Generic User Guide

Release Information

Document history

Issue	Date	Confidentiality	Change
0002-00	11 September 2017	Non-Confidential	First release for r0p2
0003-00	28 November 2017	Non-Confidential	First release for r0p3
0004-00	10 April 2018	Non-Confidential	First release for r0p4
0100-01	19 June 2020	Non-Confidential	First release for r1p0
0100-05	15 January 2023	Non-Confidential	Second release for r1p0

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

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.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Product revision status

The r*x*p*y* identifier indicates the revision status of the product described in this manual, for example, r1p2, where:

r*x* Identifies the major revision of the product, for example, r1.
p*y* Identifies the minor revision or modification status of the product, for example, p2.

1.2 Intended audience

This book is written for application and system-level software developers, familiar with programming, who want to program a device that includes the Cortex®-M33 processor.

1.3 Conventions

The following subsections describe conventions used in Arm documents.

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: developer.arm.com/glossary.

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<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>`

Convention	Use
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Recommendations. Not following these recommendations might lead to system failure or damage.

Requirements for the system. Not following these requirements might result in system failure or damage.

Requirements for the system. Not following these requirements will result in system failure or damage.

An important piece of information that needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

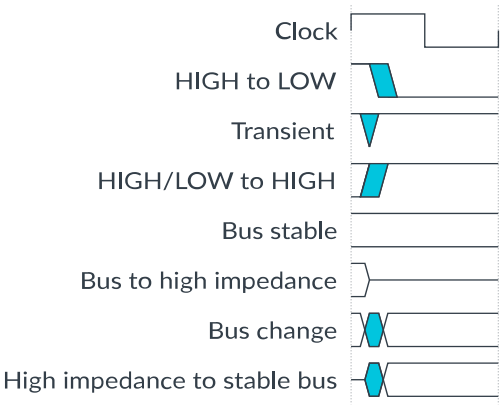
A reminder of something important that relates to the information you are reading.

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 1-1: Key to timing diagram conventions



Signals

The signal conventions are:

Signal level

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

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

Lowercase n

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

1.4 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Cortex®-M33 Processor Integration and Implementation Manual	100323	Confidential
CoreSight™ Components Technical Reference Manual	DDI 0314	Non-Confidential
Lazy Stacking and Context Switching Application Note 298	DAI0298	Non-Confidential

Arm architecture and specifications	Document ID	Confidentiality
ACLE Extensions for Arm®v8-M	100739	Non-Confidential

Arm architecture and specifications	Document ID	Confidentiality
AMBA® APB Protocol Specification	IHI 0024	Non-Confidential
AMBA® ATB Protocol Specification	IHI 0032	Non-Confidential
AMBA® Low Power Interface Specification	IHI 0068	Non-Confidential
Arm® AMBA® 5 AHB Protocol Specification	IHI 0033	Non-Confidential
Arm® CoreSight™ Architecture Specification v3.0	IHI 0029	Non-Confidential
Arm® Debug Interface Architecture Specification, ADIv5.0 to ADIv5.2	IHI 0031	Non-Confidential
Arm® Embedded Trace Macrocell Architecture Specification ETMv4	IHI 0064	Non-Confidential
Arm® Synchronization Primitives Development Article	ID012816	Non-Confidential
Arm®v8-M Architecture Reference Manual	DDI 0553	Non-Confidential
Arm®v8-M Exception Handling	100701	Non-Confidential
Arm®v8-M Processor Debug	100734	Non-Confidential
Fault Handling and Detection	100691	Non-Confidential
Introduction to the Arm®v8-M Architecture	100688	Non-Confidential
Memory Protection Unit for Arm®v8-M based platforms	100699	Non-Confidential
TrustZone® technology for Arm®v8-M Architecture	100690	Non-Confidential

Non-Arm resources	Document ID	Organization
IEEE Std 1149.1-2001, *Test Access Port and Boundary-Scan Architecture (JTAG).*	IEEE 1149.1-2001	IEEE www.ieee.org
ANSI/IEEE Std 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic.*	IEEE 754-2008	IEEE www.ieee.org

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com.

2. Overview

This chapter introduces the Cortex®-M33 processor and its features.

2.1 About the Cortex®-M33 processor and core peripherals

The Cortex®-M33 processor is a high-performance 32-bit processor that is designed for the microcontroller market. The processor offers outstanding performance, 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 signal processing applications.
- Ultra-low power consumption with integrated sleep modes.
- Platform robustness with optional integrated memory protection.
- Extended security features with optional Security Extension for Arm®v8-M.

Processor implementation

The Cortex®-M33 processor is built on a high-performance processor core, with a 3-stage pipeline Harvard architecture, making it ideal for demanding embedded applications. The in-order processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design.

The Cortex®-M33 processor provides high-end processing hardware including:

- IEEE754-compliant single-precision floating-point computation.
- *Single Instruction Multiple Data* (SIMD) multiplication and multiply-with-accumulate capabilities.
- Saturating arithmetic and dedicated hardware division.

Figure 2-1: Cortex®-M33 processor implementation without the Security Extension

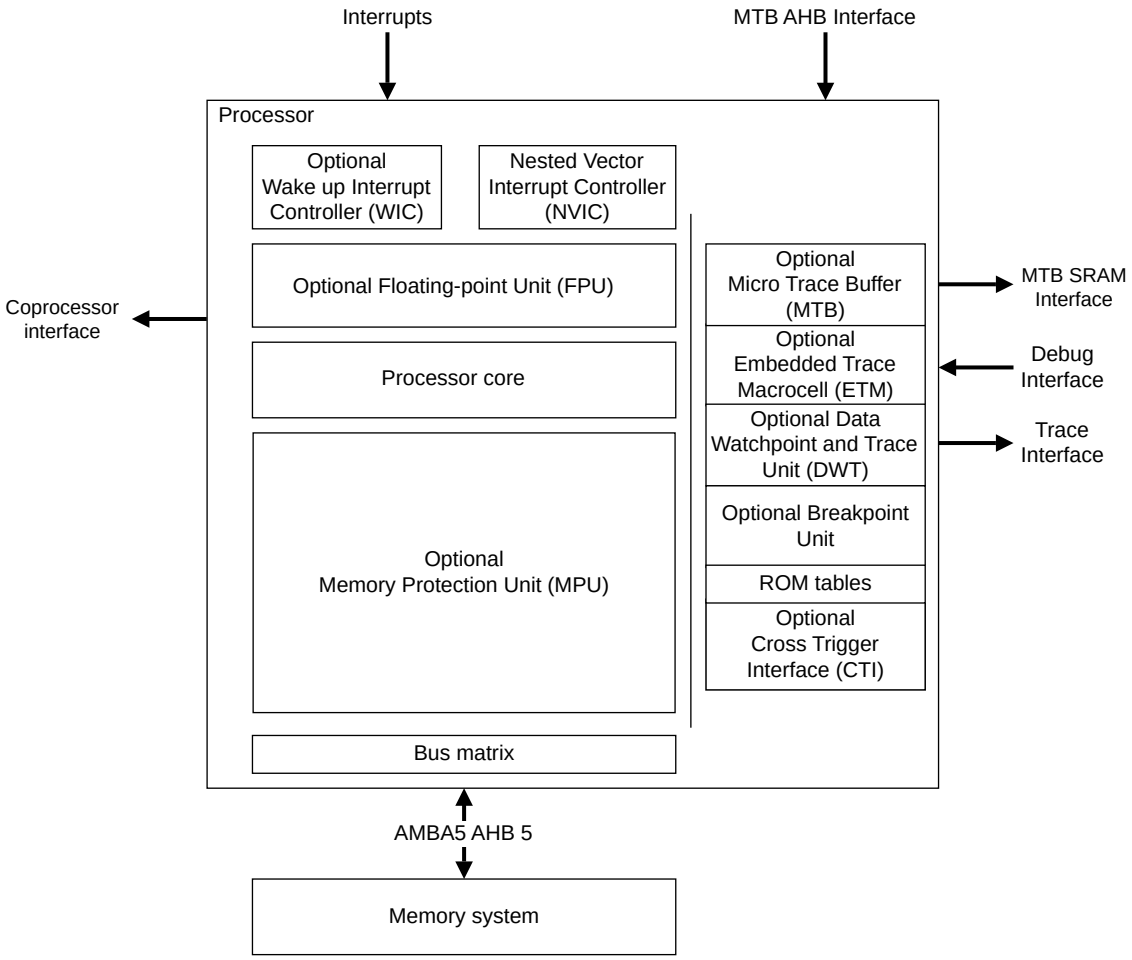
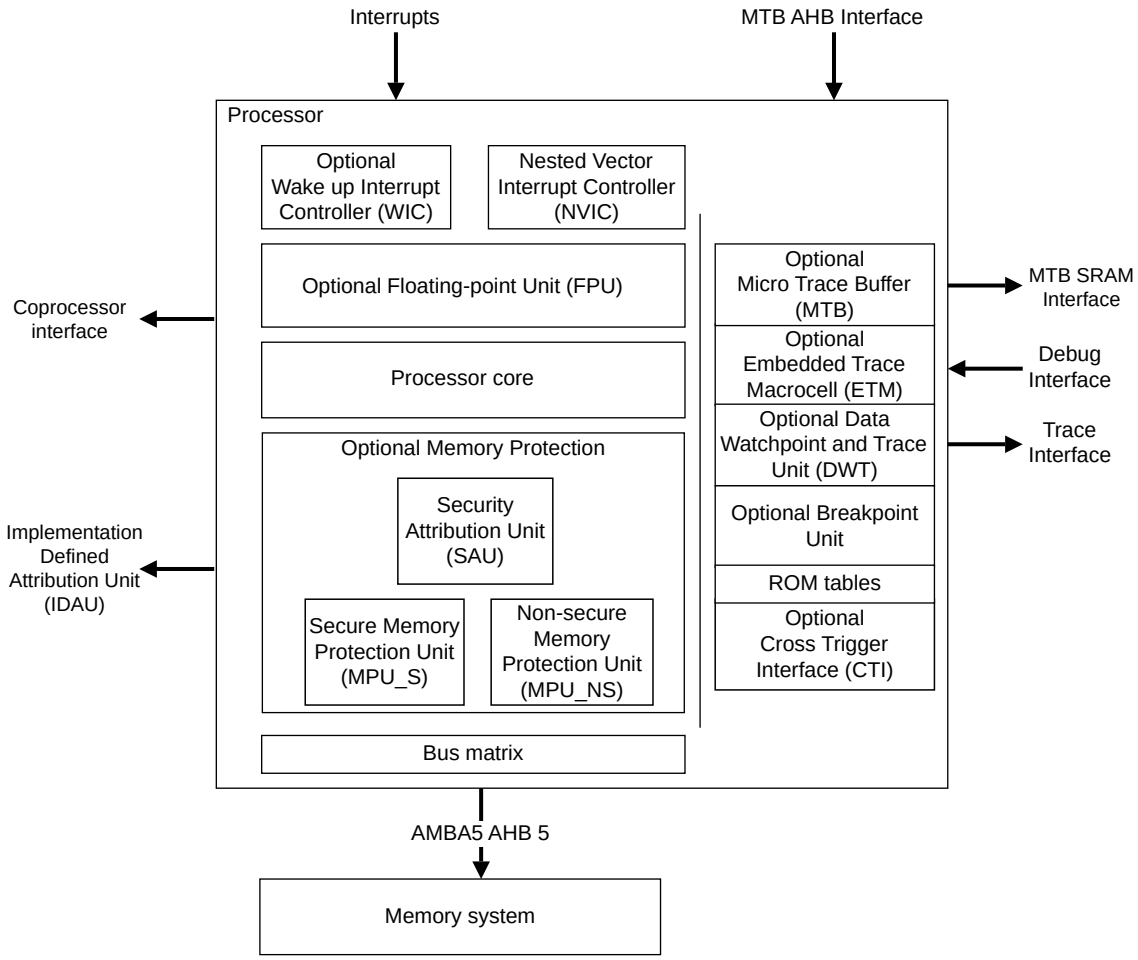


Figure 2-2: Cortex®-M33 processor implementation with the Security Extension



To facilitate the design of cost-sensitive devices, the Cortex®-M33 processor implements tightly-coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex®-M33 processor implements the T32 instruction set based on Thumb®-2 technology, ensuring high code density and reduced program memory requirements. The Cortex®-M33 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®-M33 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.

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®-M33 processor can execute certain pairs of 16-bit instructions simultaneously. This is called dual issue.

Related information

Exception entry and return on page 61

2.1.1 System-level interface

The Cortex®-M33 processor provides multiple interfaces using Arm® AMBA® technology to provide high speed, low latency memory accesses.

2.1.2 Security Extension

The Arm®v8-M 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:

- 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.
- Secure memory can only be accessed from Secure state.

2.1.3 Integrated configurable debug

The Cortex®-M33 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.

2.1.4 Processor features and benefits summary

The Cortex®-M33 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 single-precision *Floating-Point Unit* (FPU).

Other processor features and benefits are:

- Power control optimization of system components.
- Integrated sleep modes for low power consumption.
- Arm®v8-M 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.
- MPU and SAU for safety-critical applications.
- Extensive debug and trace capabilities.

2.1.5 Processor core peripherals

The processor has the following core peripherals:

Nested Vectored Interrupt Controller

The NVIC is an embedded interrupt controller that supports low-latency interrupt processing.

System Control Space

The 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.

Security Attribution Unit

The SAU improves system security by defining security attributes for different regions. It provides up to eight different regions and a default background region.

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.

Floating-point Unit

The *Floating-point Unit* (FPU) provides IEEE754-compliant operations on 32-bit single-precision floating-point values.

2.2 Arm®v8-M enablement

The following list of documents, while not specific to this product, contain important information that can assist you in developing your Cortex®-M33 processor.

Arm useful resources	Document ID	Confidentiality
ACLE Extensions for Arm®v8-M	100739	Non-Confidential
Arm® Synchronization Primitives Development Article (ID012816)	ID012816	Non-Confidential
Arm®v8-M Architecture Reference Manual	DDI 0553	Non-Confidential
Arm®v8-M Exception Handling	100701	Non-Confidential
Arm®v8-M Processor Debug	100734	Non-Confidential
Fault Handling and Detection	100691	Non-Confidential
Introduction to the Arm®v8-M Architecture	100688	Non-Confidential
Memory Protection Unit for Arm®v8-M based platforms	100699	Non-Confidential
TrustZone® technology for Arm®v8-M Architecture	100690	Non-Confidential

3. The Cortex®-M33 Processor

This chapter describes how to program the Cortex®-M33 processor.

3.1 Programmer's model

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

3.1.1 Processor modes and privilege levels for software execution

Descriptions of the two *modes* and two *privilege levels* available are provided in this topic.

Modes

Thread mode

Intended for applications.

The processor enters Thread mode out of reset and returns to Thread mode on completion of an exception handler.

Handler mode

Intended for OS execution.

All exceptions cause entry into Handler mode.

Privilege levels

There are two levels of privilege:

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 register 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.

3.1.2 Security states

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.

3.1.3 Core registers

The following figures and tables illustrate the core registers of the Cortex®-M33 processor:

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

Figure 3-1: Core registers without the Security Extension

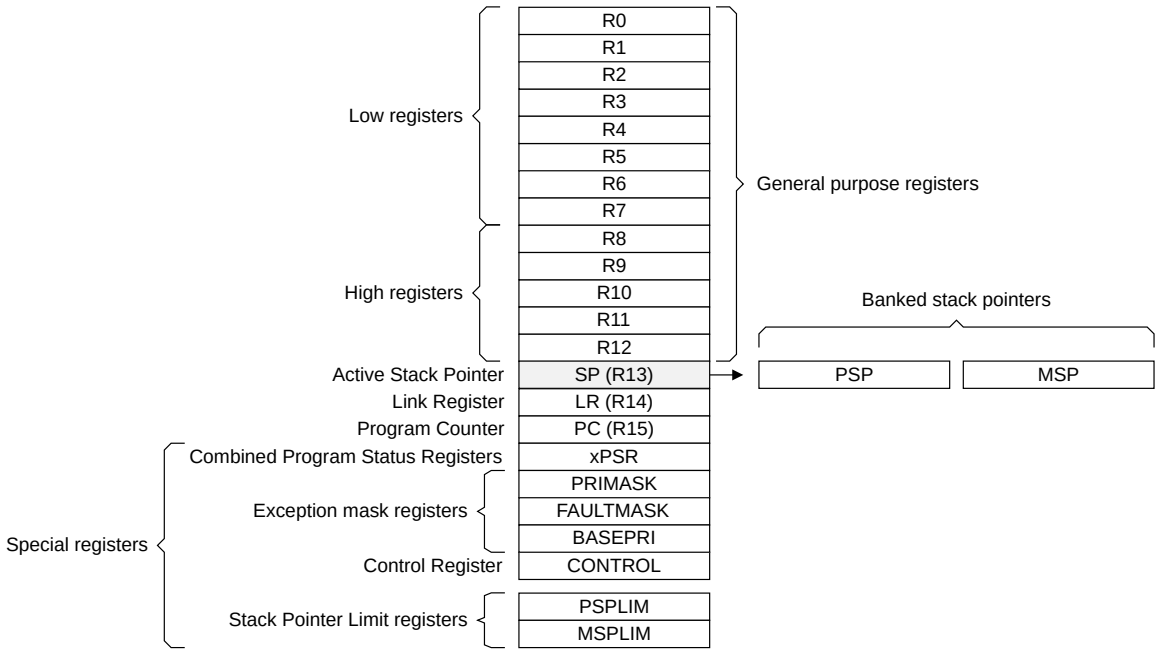
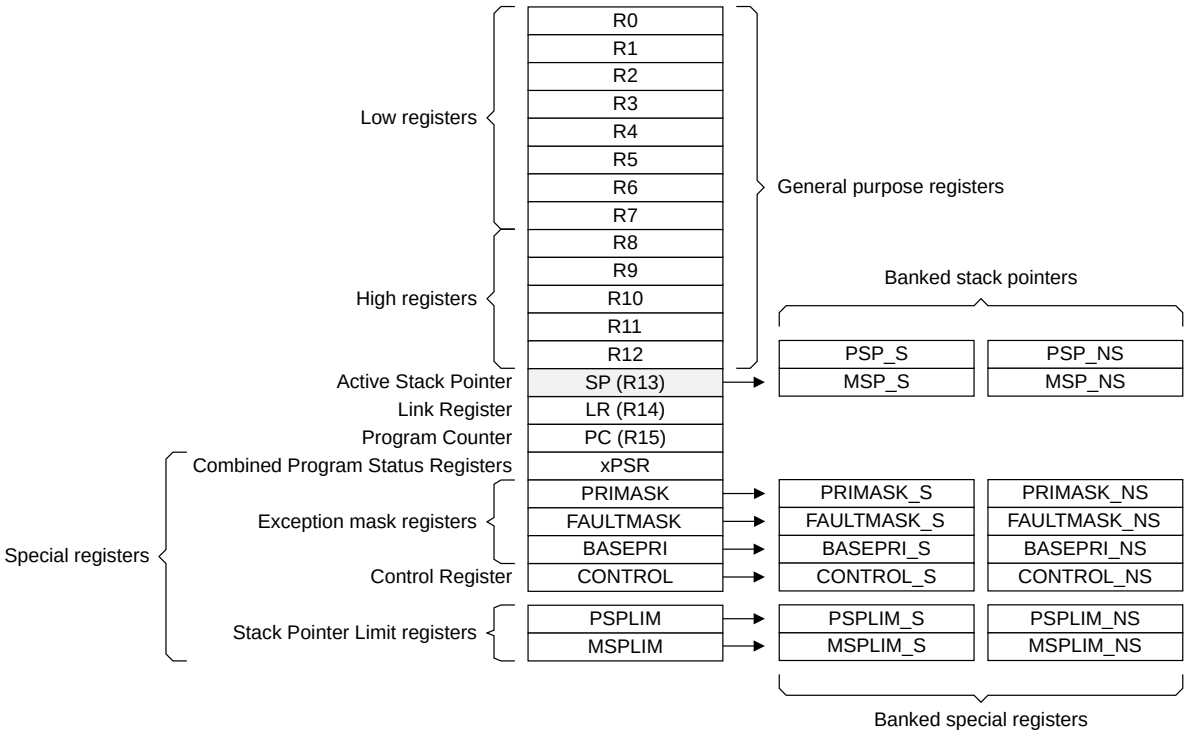


Table 3-1: Core register set summary without the Security Extension

Name	Type[1]	Required privilege[2]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	3.1.3.1 General-purpose registers on page 27
MSP	RW	Either	-[3]	3.1.3.2 Stack Pointer on page 27
PSP	RW	Either	UNKNOWN	
LR	RW	Either	0xFFFFFFFF	3.1.3.4 Link Register on page 29
PC	RW	Either	-[3]	3.1.3.5 Program Counter on page 29

Name	Type[1]	Required privilege[2]	Reset value	Description
xPSR (includes APSR, IPSR, and EPSR)	RW	Either	-[4]	3.1.3.6 Combined Program Status Register on page 30
APSR	RW	Either	UNKNOWN	3.1.3.6.1 Application Program Status Register on page 30
IPSR	RO	Privileged	0x00000000	3.1.3.6.2 Interrupt Program Status Register on page 31
EPSR	RO	Privileged	-[4]	3.1.3.6.3 Execution Program Status Register on page 32.
PRIMASK	RW	Privileged	0x00000000	3.1.3.7.1 Priority Mask Register on page 34
FAULTMASK	RW	Privileged	0x00000000	3.1.3.7.2 Fault Mask Register on page 35
BASEPRI	RW	Privileged	0x00000000	3.1.3.7.3 Base Priority Mask Register on page 37
CONTROL	RW	Privileged	0x00000000	3.1.3.8 CONTROL register on page 38
PSPLIM	RW	Privileged	0x00000000	3.1.3.3 Stack limit registers on page 28
MSPLIM	RW	Privileged		

Figure 3-2: Core registers with the Security Extension



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

Table 3-2: Core register set summary with the Security Extension

Name	Type[1]	Required privilege[2]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	3.1.3.1 General-purpose registers on page 27.
MSP_S	RW	Either	-[3]	3.1.3.2 Stack Pointer on page 27
MSP_NS		Either		
PSP_S	RW	Either	UNKNOWN	
PSP_NS		Either		
LR	RW	Either	UNKNOWN	3.1.3.4 Link Register on page 29
PC	RW	Either	-[3]	3.1.3.5 Program Counter on page 29
xPSR (includes APSR, IPSR, and EPSR)	RW	Either	-[4]	3.1.3.6 Combined Program Status Register on page 30
APSR	RW	Either	UNKNOWN	3.1.3.6.1 Application Program Status Register on page 30.
IPSR	RO	Privileged	0x00000000	3.1.3.6.2 Interrupt Program Status Register on page 31
EPSR	RO	Privileged	-[4]	3.1.3.6.3 Execution Program Status Register on page 32
PRIMASK_S	RW	Privileged	0x00000000	3.1.3.7.1 Priority Mask Register on page 34
PRIMASK_NS		Privileged	0x00000000	
FAULTMASK_S	RW	Privileged	0x00000000	3.1.3.7.2 Fault Mask Register on page 35
FAULTMASK_NS		Privileged	0x00000000	
BASEPRI_S	RW	Privileged	0x00000000	3.1.3.7.3 Base Priority Mask Register on page 37
BASEPRI_NS		Privileged	0x00000000	
CONTROL_S	RW	Privileged	0x00000000	3.1.3.8 CONTROL register on page 38
CONTROL_NS		Privileged	0x00000000	
MSPLIM_S	RW	Privileged	0x00000000	3.1.3.3 Stack limit registers on page 28
MSPLIM_NS		Privileged	0x00000000	
PSPLIM_S	RW	Privileged	0x00000000	
PSPLIM_NS		Privileged	0x00000000	

3.1.3.1 General-purpose registers

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

3.1.3.2 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.

When Security state is implemented, software must initialize MSP_NS.

Table 3-3: 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 3-4: 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.

3.1.3.3 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®-M33 processor has two stack limit registers, as the following table shows.

Table 3-5: Stack limit registers without the Security Extension

Stack	Stack limit register
Main	MSPLIM
Process	PSPLIM

If the Security Extension is implemented, the Cortex®-M33 processor has four stack limit registers, as the following table shows.

Table 3-6: 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

The four stack limit registers are banked between Security states.

See Table 3-1: Core register set summary without the Security Extension on page 25 table for the stack limit registers attributes.

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

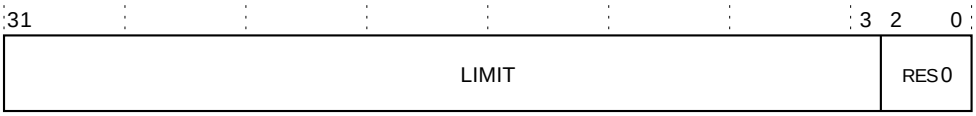


Table 3-7: MSPLIM and PSPLIM register bit assignments

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**.

3.1.3.4 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`.

3.1.3.5 Program Counter

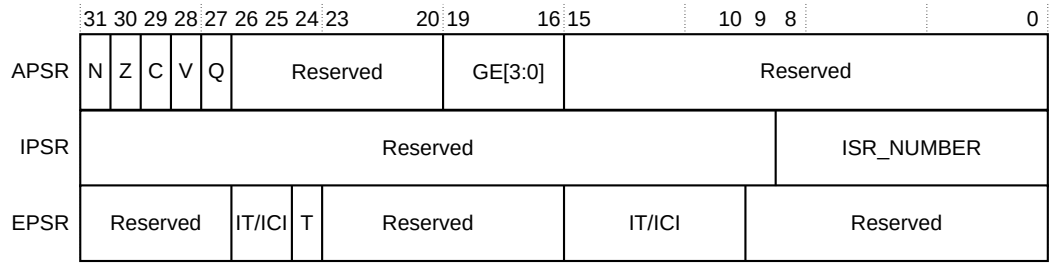
The *Program Counter* (PC) is register R15. It contains the current program address.

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

3.1.3.6 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:



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 3-8: xPSR register combinations

Register	Type	Combination
xPSR	RW[5, 6]	APSR, EPSR, and IPSR
IEPSR	RO[6]	EPSR and IPSR
IAPSR	RW[5]	APSR and IPSR
EAPSR	RW[6]	APSR and EPSR

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

3.1.3.6.1 Application Program Status Register

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

See Table 3-1: Core register set summary without the Security Extension on page 25 for the APSR attributes.

The APSR bit assignments are as follows:

[5] The processor ignores writes to the IPSR bits.
[6] Reads of the EPSR bits return zero, and the processor ignores writes to these bits.

Table 3-9: 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 4.4.12 SEL on page 116 for more information.
[15:0]	-	Reserved.

3.1.3.6.2 Interrupt Program Status Register

The IPSR contains the exception number of the current ISR.

The bit assignments are:

Table 3-10: IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved.

Bits	Name	Function
[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].

3.1.3.6.3 Execution Program Status Register

The EPSR contains the Thumb® state bit and the execution state bits for the *If-Then* (IT) instruction, and *Interruptible-Continuable Instruction* (ICI) field for an interrupted load multiple or store multiple instruction.

See the Table 3-1: Core register set summary without the Security Extension on page 25 for the EPSR attributes.

The following table shows the EPSR bit assignments.

Table 3-11: EPSR bit assignments

Bits	Name	Function
[31:27]	-	Reserved
[26:25], [15:10]	ICI	Interruptible-continuable instruction bits, see 3.1.3.6.4 Interruptible-continuable instructions on page 33
[26:25], [15:10]	IT	Indicates the execution state bits of the `IT` instruction, see 4.11.5 IT on page 175
[24]	T	Thumb® state bit, see 3.1.3.6.6 Thumb state on page 34
[23:16]	-	Reserved
[9:0]	-	Reserved

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

3.1.3.6.4 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.

3.1.3.6.5 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.

3.1.3.6.6 Thumb state

The Cortex®-M33 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 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 3.5.4 Lockup on page 73 for more information.

3.1.3.7 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.

3.1.3.7.1 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.

See Table 3-1: Core register set summary without the Security Extension on page 25 table for the PRIMASK attributes.

The bit assignments for the PRIMASK register are as follows:

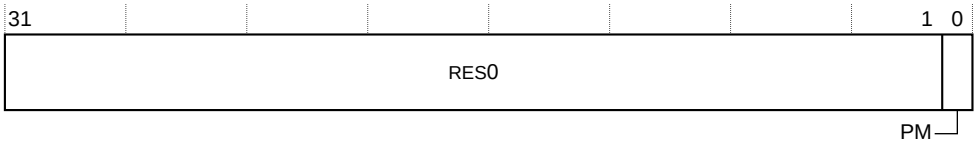


Table 3-12: PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, **RES0**.
[0]	PM	Setting this bit to one boosts the current execution priority to 0, masking all exceptions with a programmable priority. Setting PRIMASK_S to one boosts the current execution priority to 0. If AIRCR.PRIS is: **0** Setting PRIMASK_NS to one boosts the current execution priority to `0x0`. **1** Setting PRIMASK_NS to one boosts the current execution priority to `0x80`. When the current execution priority is boosted to a particular value, all exceptions with a lower or equal priority are masked.

Table 3-13: PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, **RES0**.
[0]	PM	In an implementation without the Security Extension, setting this bit to one boosts the current execution priority to 0, masking all exceptions with a programmable priority. In an implementation with the Security Extension, setting PRIMASK_S to one boosts the current execution priority to 0. If AIRCR.PRIS is: **0** Setting PRIMASK_NS to one boosts the current execution priority to `0x0`. **1** Setting PRIMASK_NS to one boosts the current execution priority to `0x80`. When the current execution priority is boosted to a particular value, all exceptions with a lower or equal priority are masked.

3.1.3.7.2 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:

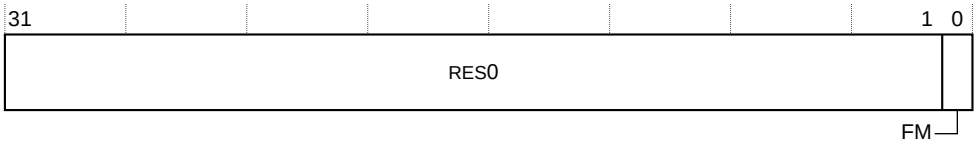


Table 3-14: FAULTMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, **RES0**
[0]	FM	Setting this bit to one boosts the current execution priority to -1, masking all exceptions except NMI. Setting this bit to one boosts the current execution priority to -1, masking all exceptions with a lower priority. 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.

Table 3-15: 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.

Table 3-16: FAULTMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved, **RES0**
[0]	FM	Setting this bit to one boosts the current execution priority to -1, masking all exceptions with a lower priority. 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.

3.1.3.7.3 Base Priority Mask Register

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

See Table 3-1: Core register set summary without the Security Extension on page 25 table for the BASEPRI register attributes.

The bit assignments for the BASEPRI register are as follows:

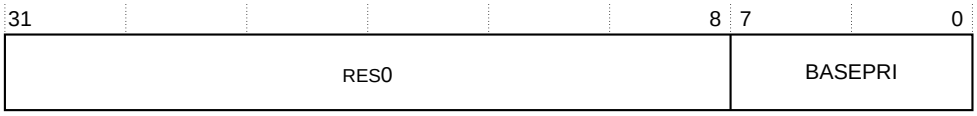


Table 3-17: BASEPRI register bit assignments

Bits	Name	Function
[31:8]	-	Reserved, **RES0**
[7:0]	BASEPRI[7]	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 are masked. Writing 0 to BASEPRI disables base priority boosting.

[7] 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. See 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279 for more information. Remember that higher priority field values correspond to lower exception priorities.

3.1.3.8 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 state is active.

See Table 3-1: Core register set summary without the Security Extension on page 25 table for the CONTROL register attributes.

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 are as follows:

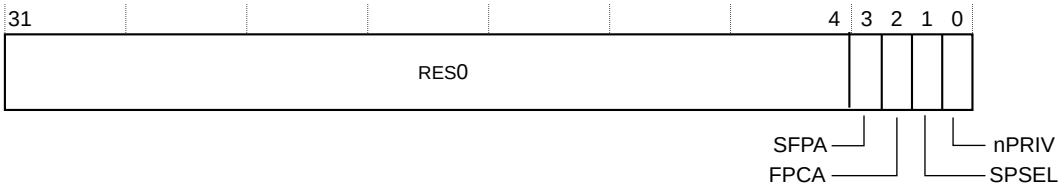


Table 3-18: CONTROL register bit assignments

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.
[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®-M33 core updates this bit automatically on exception return. This bit is banked between Security states.

Bits	Name	Function
[0]	nPRIV	Defines the Thread mode privilege level: **0** Privileged. **1** Unprivileged. This bit is banked between Security states.

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.

3.1.4 Exceptions and interrupts

The Cortex®-M33 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, handled in Handler mode. Exceptions can target either Security state.

The NVIC registers control interrupt handling.

Related information

Nested Vectored Interrupt Controller on page 272

3.1.5 Data types and data memory accesses

The Cortex®-M33 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.
- 32-bit single-precision floating-point numbers.
- 64-bit double-precision floating-point numbers.

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

For a Cortex®-M33 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.

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex®-M33 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.

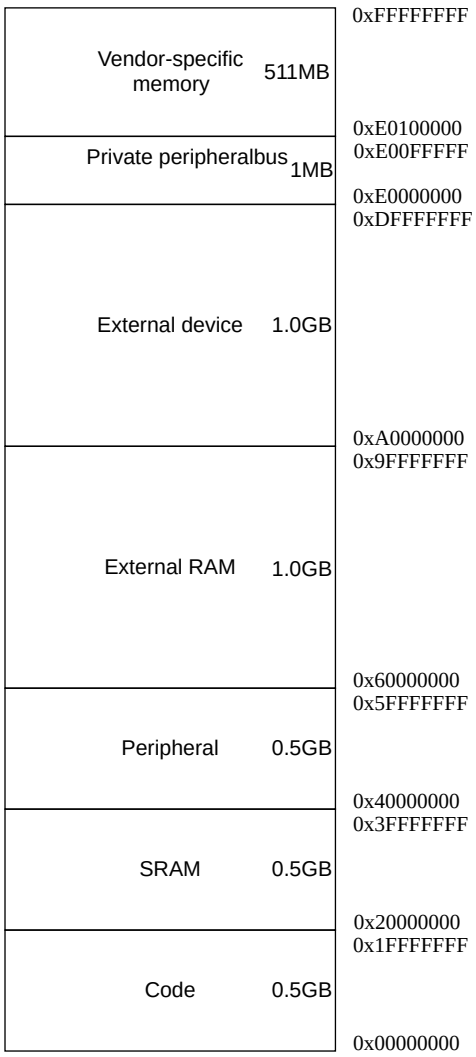
3.2 Memory model

The Cortex®-M33 processor has a fixed default memory map that provides up to 4GB of addressable memory.

3.2.1 Processor memory map

The Cortex®-M33 processor memory map.

Figure 3-3: Cortex®-M33 processor memory map



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

3.2.2 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.

The additional memory attributes include:

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)
: Means that the processor prevents instruction accesses. A MemManage fault exception is generated on executing an instruction fetched from an XN region of memory.

3.2.3 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®-M33 processor, only two combinations of these attributes are valid:

- Device-nGnRnE.
- Device-nGnRE.

Note

- Device-nGnRnE is equivalent to Arm®v7-M Strongly Ordered memory type
- Device-nGnRE is equivalent to Arm®v7-M Device memory.
- Device-nGRE and Device-GRE are new to the Arm®v8-M architecture.

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

Note

Device memory is shareable, and must not be cached.

3.2.4 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.

Note

The partitioning of the memory into Secure and Non-secure regions is independent of the Security state that the processor executes in. See 3.4 Security state switches on page 68 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 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 encoding values, 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 IDAU are marked as Non-secure on the AHB bus.

The MPU is banked between Secure and Non-secure memory. For instructions 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 data stores, accesses depend on the Security state of the processor. For example, if the processor is in Secure state the access is subject to the Secure MPU settings. If the processor is in Non-secure state the access is subject to the Non-secure MPU settings.

3.2.5 Behavior of memory accesses

Summary of the behavior of accesses to each region in the memory map.

Table 3-19: 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 3.2.2 Memory regions, types, and attributes on page 41.

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

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

3.2.5.1 Additional memory access constraints for caches and shared memory

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

This behavior is shown by the following table:

Table 3-20: Memory region shareability and cache policies

Address range	Memory region	Memory type	Shareability	Cache policy
0x00000000-0x1FFFFFFF	Code	Normal	-	WT[8]
0x20000000-0x3FFFFFFF	SRAM	Normal	-	WBWA[9]
0x40000000-0x5FFFFFFF	Peripheral	Device	Shareable	-
0x60000000-0x7FFFFFFF	RAM	Normal	-	WBWA[9]
0x80000000-0x9FFFFFFF				WT[8]
0xA0000000-0xDFFFFFFF	External device	Device	Shareable	-
0xE0000000-0xE003FFFF	Private Peripheral Bus	Device	Shareable	-
0xE0040000-0xE0043FFF	Device	Device	Shareable	-
0xE0044000-0xE00EFFFF	Private Peripheral Bus	-	Shareable	Device
0xF0000000-0xFFFFFFFF	Vendor_SYS	Device	Shareable	Device


Note

For more information on memory types and shareability, see 3.2.2 Memory regions, types, and attributes on page 41.

3.2.6 Software ordering of memory accesses

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions.

In the Cortex®-M33 processor this behavior can occur because of two reasons:

- Memory or devices in the memory map might have different wait states.
- Some memory accesses associated with instruction fetches are speculative.

3.2.3 Device memory on page 42 describes the cases where the memory system guarantees the order of memory accesses. Otherwise, if the order of memory accesses is critical, software must include memory barrier instructions to force that ordering.

[8] WT means Write through, no write allocate.
[9] WBWA means Write back, write allocate.

The processor provides the following memory barrier instructions:

DMB The *Data Memory Barrier* (DMB) instruction ensures that outstanding memory transactions complete before subsequent memory transactions.

DSB The *Data Synchronization Barrier* (DSB) instruction ensures that outstanding memory transactions complete before subsequent instructions execute.

ISB The *Instruction Synchronization Barrier* (ISB) ensures that the effect of any context-changing operations is recognizable by subsequent instructions.

The following are examples of using memory barrier instructions:

Exception vector and vector table programming

If the program changes an entry in the vector table, and then enables the corresponding exception, use a `DMB` instruction between the operations. This ensures that if the exception is taken immediately after being enabled, then the processor uses the new exception vector.

If the program updates the value of the VTOR, use a `DMB` instruction to ensure that the new vector table is used for subsequent exceptions.

Self-modifying code

If a program contains self-modifying code, use a `DSB` instruction followed by an `ISB` instruction immediately after the code modification in the program. This ensures subsequent instruction execution uses the updated program.

Memory map switching

If the system contains a memory map switching mechanism, use a `DSB` instruction followed by an `ISB` instruction after switching the memory map. This ensures subsequent instruction execution uses the updated memory map.

MPU programming

Use a `DSB` followed by an `ISB` instruction or exception return to ensure that the new MPU configuration is used by subsequent instructions.

SAU programming

Use a `DSB` followed by an `ISB` instruction or exception return to ensure that the SAU configuration is used by subsequent instructions.

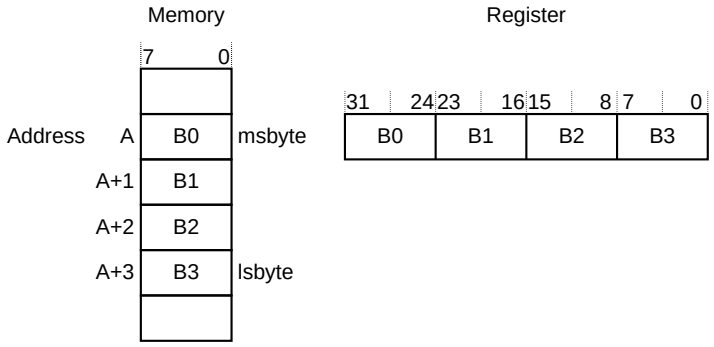
3.2.7 Memory endianness

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.

3.2.7.1 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.

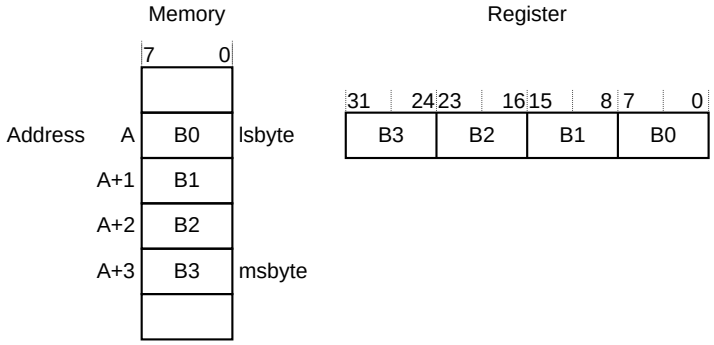
Example 3-1: Byte-invariant big-endian example



3.2.7.2 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.

Example 3-2: Little-endian example



3.2.8 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`.

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 in the system.

For more information about the synchronization primitive instructions, see 4.14.11 LDREX and STREX on page 232 and 4.14.13 CLREX on page 235.

A global exclusive access can be performed:

- In a Shared region if the MPU is implemented.
- By setting ACTLR.EXTEXCLALL. In this case, exclusive information is always sent externally.

In any other case, exclusive information is not sent on the AHB bus, HEXCL is 0, and only the local monitor is used.

If HEXCL is sent externally and there is no exclusive monitor for the corresponding memory region, then STREX and STLEX fails.

3.2.9 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.

Table 3-21: CMSIS functions for exclusive access instructions

Instruction	CMSIS function
LDAEX	uint16_t __LDAEX (volatile uint16_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	uint16_t __STLEX (uint16_t value, volatile uint16_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
```

3.3 Exception model

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

3.3.1 Exception states

Each exception is in one 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.


Note

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.

3.3.2 Exception types

This section describes the exception types for a processor with and without the Security Extension.

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 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.

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.

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.

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.

Table 3-22: 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
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

Exception number (see notes)	IRQ number (see notes)	Exception type	Priority	Vector address	Activation
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 3.1.3.6.2 Interrupt Program Status Register on page 31.
- For configurable priority values, see 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279.

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.

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

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.

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.

Table 3-23: 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

- 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 3.1.3.6.2 Interrupt Program Status Register on page 31.
- For configurable priority values, see 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279.

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.

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

3.3.3 Exception handlers

The exception handlers are the following:

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 in Secure or Non-secure state, using NVIC_ITNS.

Fault handler

The fault handler handles 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.

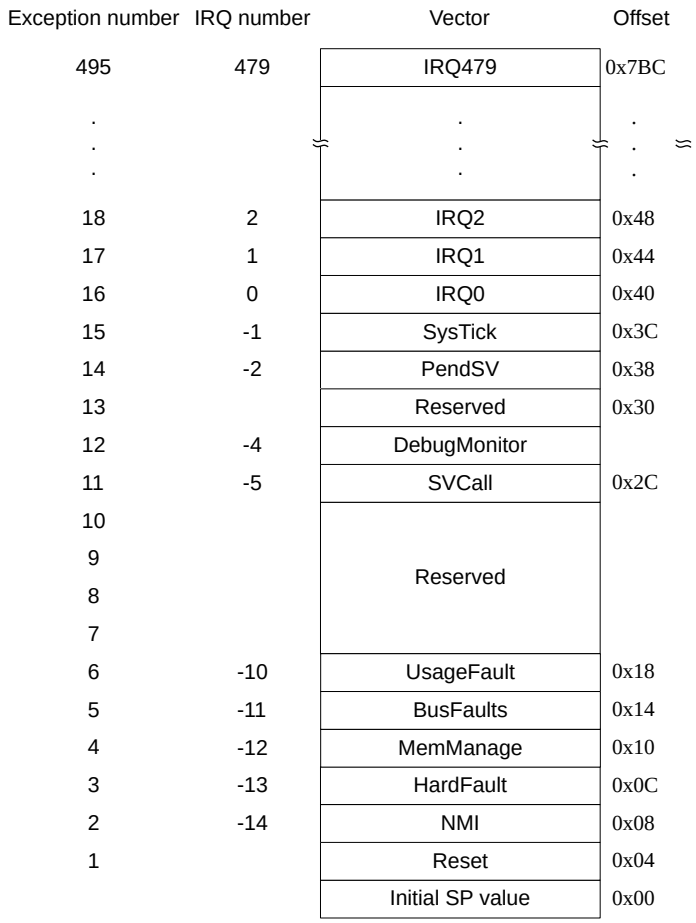
3.3.4 Vector table

The *Vector Table Offset Register* (VTOR) 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.

Figure 3-4: 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.

Figure 3-5: Vector table with the Security Extension

Exception number	IRQ number	Secure Vector	Non-secure Vector	Offset
495	479	IRQ479	IRQ479	0x7BC
. . .			. . .	. . .
18	2	IRQ2	IRQ2	0x48
17	1	IRQ1	IRQ1	0x44
16	0	IRQ0	IRQ0	0x40
15	-1	SysTick_S	SysTick_NS	0x3C
14	-2	PendSV_S	PendSV_NS	0x38
13		Reserved	Reserved	0x30
12	-3	DebugMonitor	DebugMonitor	
11	-5	SVCall_S	SVCall_NS	0x2C
10		Reserved	Reserved	
9				
8				
7	-9	SecureFault		0x1C
6	-10	UsageFault_S	UsageFault_NS	0x18
5	-11	BusFault_S	BusFault_NS	0x14
4	-12	MemManage_S	MemManage_NS	0x10
3	-13	HardFault_S	HardFault_NS	0x0C
2	-14	NMI_S	NMI_NS	0x08
1		Reset		0x04
		Initial SP value		0x00

Because reset always targets Secure state, the Non-secure Reset and Non-secure Initial SP value are ignored by the hardware.

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.

3.3.5 Exception priorities

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

If software does not configure any priorities, then all exceptions with a configurable priority have a priority of 0. For information about configuring exception priorities, see:

- 5.2.9 M33 System Handler Priority Registers on page 253.
- 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279.

Configurable priorities 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 exceptions, 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.

Table 3-24: 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.

3.3.6 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.

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.

3.3.7 Exception entry and return

Descriptions of exception handling use the following terms.

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.

3.3.7.1 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*.

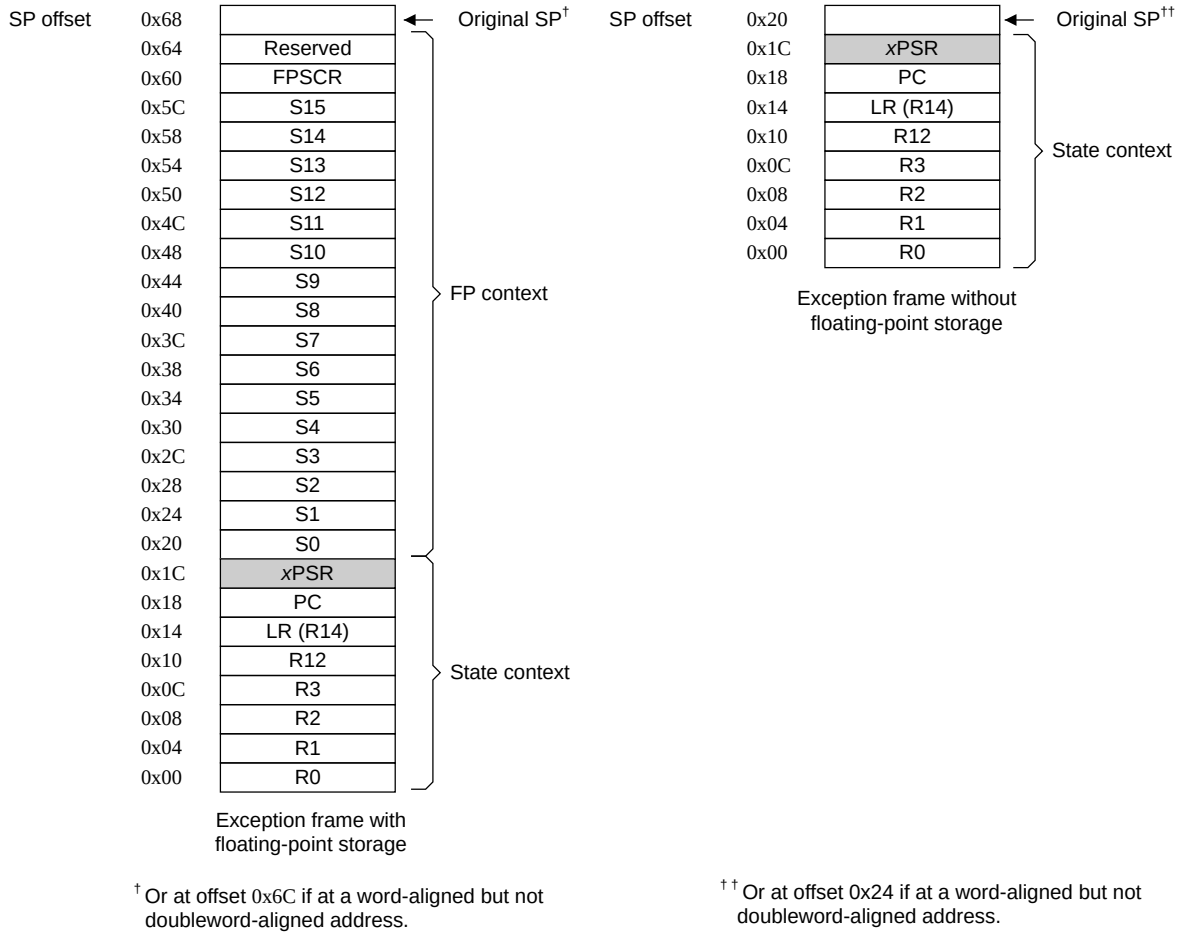
If the floating-point context is active, the Cortex®-M33 processor can automatically stack the architected floating-point state on exception entry. The following figure shows the Cortex®-M33 processor stack frame layout when an interrupt or an exception is preserved on the stack:

- with floating-point state.
- without floating-point state.


Note

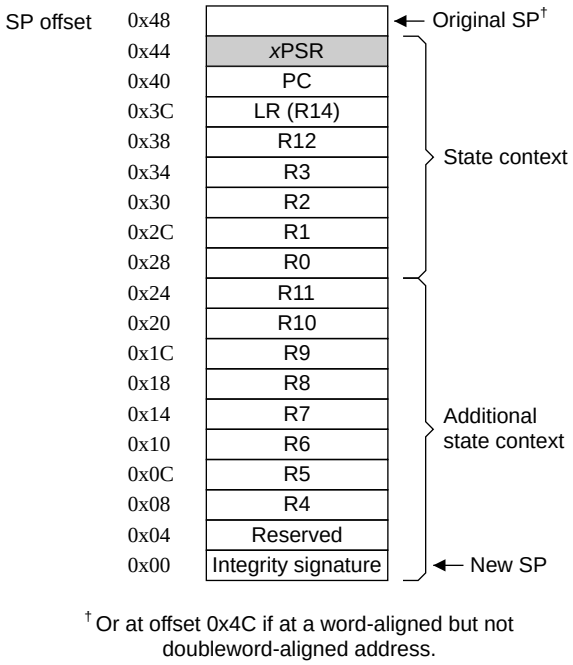
Where stack space for floating-point state is not allocated, the stack frame is the same as that of Arm®v8-M implementations without an FPU.

Figure 3-6: 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.

Figure 3-7: Stack frame extended to save additional context when the Security Extension is implemented



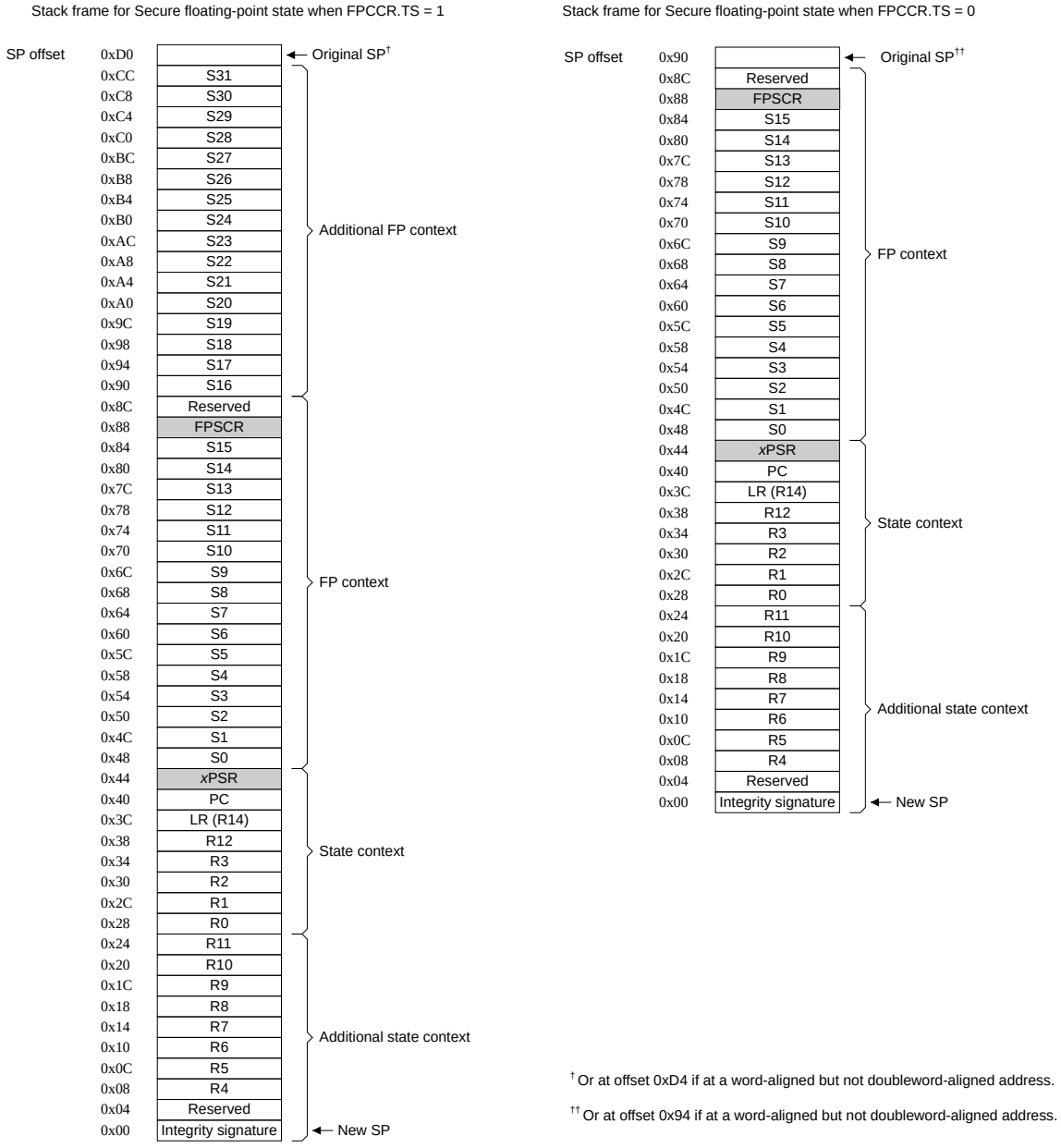
† Or at offset 0x4C if at a word-aligned but not doubleword-aligned address.

If the floating-point context is active, the Cortex®-M33 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.

Figure 3-8: 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 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.

3.3.7.2 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 3-25: Exception return behavior

Bits	Name	Function
[31:24]	PREFIX	Indicates that this is an EXC_RETURN value. This field reads as `0b11111111`.
[23:7]	-	Reserved, **RES1**.
[6]	S	Indicates whether registers have been pushed to a Secure or Non-secure stack. **0** Non-secure stack used. **1** Secure stack used.

Bits	Name	Function
[5]	DCRS	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.
[2]	SPSEL	Indicates the transitory value of the CONTROL.SPSEL bit associated with the Security state of the exception as indicated by EXC_RETURN.ES. **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 3-26: Exception return behavior

Bits	Name	Function
[31:24]	PREFIX	Indicates that this is an EXC_RETURN value. This field reads as `0b11111111`.
[23:7]	-	Reserved, **RES1**.
[6]	-	Reserved, **RES0**.
[5]	-	Reserved, **RES1**.
[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**.

Bits	Name	Function
[3]	Mode	Indicates the mode that was stacked from. **0** Handler mode. **1** Thread mode.
[2]	SPSEL	Indicates which stack contains the exception stack frame. **0** Main stack pointer. **1** Process stack pointer.
[1:0]	-	Reserved, **RES0**.

3.4 Security state switches

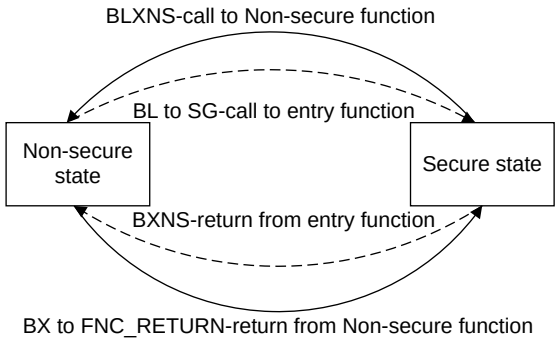
The following table presents the possible security transitions, the instructions that can cause them, and any faults that may be generated.

Table 3-27: 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 3-9: 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.

3.5 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 are generated by:

- A bus error on:
  - An instruction fetch or vector table load.
  - A data access.
- An internally-detected error such as an undefined instruction.
- Attempting to execute an instruction from a memory region marked as *Execute Never* (XN).
- A privilege violation or an attempt to access an unmanaged region causing an MPU fault.
- A security violation.

3.5.1 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 3-28: Faults

Fault	Handler	Bit name	Fault status register
Bus error on a vector read	HardFault	VECTTBL	5.2.12 M33 HardFault Status Register on page 264
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	-

Fault	Handler	Bit name	Fault status register
On instruction access		IACCVIOL [10]	5.2.11.1 M33 MemManage Fault Status Register on page 259
On data access		DACCVIOL	
During exception stacking		MSTKERR	
During exception unstacking		MUNSKERR	
During lazy floating-point state preservation		MLSPERR	
Bus error:	BusFault	-	-
During exception stacking		STKERR	5.2.11.2 M33 BusFault Status Register on page 261
During exception unstacking		UNSTKERR	
During instruction prefetch		IBUSERR	
During lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	UsageFault	NOCP	5.2.11.3 UsageFault Status Register on page 262
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state [11]		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	5.5.7 Secure Fault Status Register on page 288
Lazy state preservation error flag		LSPERR	
Invalid transition flag		INVTRAN	
Attribution unit violation flag		AUVIOL	
Invalid exception return flag		INVER	
Invalid integrity signature flag		INVIS	
Invalid entry point		INVEP	

3.5.2 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.

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

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 causes the same kind of fault 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 handler causes a fault with the same or lower priority as the fault it is servicing. This is because the handler for the new fault cannot preempt the currently executing fault handler.
- An exception handler causes a fault for which the priority is the same as or lower than the currently executing exception.
- 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 0-255, 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.


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.

3.5.3 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.

For each physical fault address register, it is only possible to report the address of one fault at a time. Each fault address register 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.

Table 3-29: Fault status and fault address registers

Handler	Status register name	Address register name	Register description
HardFault	HFSR	-	5.2.12 M33 HardFault Status Register on page 264
MemManage	MMFSR[12]	MMFAR[12]	5.2.11.1 M33 MemManage Fault Status Register on page 259 5.2.13 M33 MemManage Fault Address Register on page 265

[12] MMFSR, MMFAR, and UFSR are banked between Security states.

Handler	Status register name	Address register name	Register description
BusFault	BFSR	BFAR	5.2.11.2 M33 BusFault Status Register on page 261 5.2.14 BusFault Address Register on page 265
UsageFault	UFSR[12]	-	5.2.11.3 UsageFault Status Register on page 262
SecureFault	SFSR	SFAR	5.5.7 Secure Fault Status Register on page 288 5.5.8 Secure Fault Address Register on page 289

3.5.4 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.

The processor remains in lockup state until either:

- It is reset.
- Preemption by a higher priority exception occurs.
- It is halted by a debugger.


Note

In an implementation with the Security Extension, if lockup state occurs from a Secure HardFault when AIRCR.BFHFNMINS is set to 1 or the NMI handler, a subsequent NMI does not cause the processor to leave lockup state.

3.6 Power management

The Cortex®-M33 processor supports modes for sleep and deep sleep that reduce power consumption. Sleep mode stops the processor clock. Deep sleep mode stops the system clock and, depending on the system-specific power-saving measures, switches off the PLL and flash memory.

The SCR.SLEEPDEEP bit selects which sleep mode is used. For more information about the sleep modes, see 5.2.7 System Control Register - Cortex-M33 on page 249

3.6.1 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.

3.6.1.1 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.

3.6.1.2 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, or a processor in the system has executed an `SEV` instruction.

3.6.1.3 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.

3.6.2 Wakeup from sleep mode

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

3.6.2.1 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.

3.6.2.2 Wakeup from WFE

Conditions which cause the processor to wakeup from WFE.

The processor wakes up if:

- It detects an exception with sufficient priority to cause exception entry.
- It detects an external event signal.
- In a multiprocessor system, another processor in the system executes an SEV instruction.

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.

3.6.3 The Wakeup Interrupt Controller

The *Wakeup Interrupt Controller* (WIC) is a peripheral that can detect an interrupt and wake the processor from deep sleep mode. The WIC is enabled only when the SLEEPDEEP bit in the SCR is set to 1.

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

When the WIC is enabled and the processor enters deep sleep mode, the power management unit in the system can power down most of the Cortex®-M33 processor. This might have the side effect of stopping the SysTick timer. When the WIC receives an interrupt, it takes several clock cycles to wakeup the processor and restore its state, before it can process the interrupt. This means interrupt latency is increased in deep sleep mode.


Note

If the processor detects a connection to a debugger, it disables the WIC.

3.6.4 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.

3.6.5 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
```

4. The Cortex®-M33 Instruction Set

This chapter describes the Cortex®-M33 instruction set. It provides general information and describes each Cortex®-M33 instruction in the functional group that they belong. All the instructions that the Cortex®-M33 processor supports are described.

4.1 Cortex®-M33 instructions

The T32 instruction set is supported by the Cortex®-M33 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.

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

Table 4-1: Cortex®-M33 instruction set summary

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
ADD, ADDW	{Rd,} Rn, #imm12	Add	-	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
ADR	Rd, label	Address to Register	-	4.14.2 ADR on page 220
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right	N,Z,C	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
B {cond}	label	Branch {conditionally}	-	4.11.2 B, BL, BX, and BLX on page 172
BFC	Rd, #lsb, #width	Bit Field Clear	-	4.10.2 BFC and BFI on page 170
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	-	4.10.2 BFC and BFI on page 170
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
BKPT	#imm8	Breakpoint	-	4.13.2 BKPT on page 207

Mnemonic	Operands	Brief description	Flags	Page
BL	label	Branch with Link	-	4.11.2 B, BL, BX, and BLX on page 172
BLX	Rm	Branch indirect with Link and Exchange	-	4.11.2 B, BL, BX, and BLX on page 172
BLXNS	Rm	Branch indirect with Link and Exchange, Non-secure	-	4.11.3 BXNS and BLXNS on page 174
BX	Rm	Branch and Exchange	-	4.11.2 B, BL, BX, and BLX on page 172
BXNS	Rm	Branch and Exchange, Non-secure	-	4.11.3 BXNS and BLXNS on page 174
CBNZ	Rn, label	Compare and Branch on Non Zero	-	4.11.4 CBZ and CBNZ on page 174
CBZ	Rn, label	Compare and Branch on Zero	-	4.11.4 CBZ and CBNZ on page 174
CDP, CDP2	{*cond*} *coproc*, #*op1*, *Rt*, *CRn*, *CRm*{, #*op2*}	Coprocessor Data Processing	-	4.5.3 CDP and CDP2 on page 132
CLREX	-	Clear Exclusive	-	4.14.13 CLREX on page 235
CLZ	Rd, Rm	Count Leading Zeros	-	4.4.5 CLZ on page 108
CMN	Rn, Op2	Compare Negative	N,Z,C,V	4.4.6 CMP and CMN on page 108
CMP	Rn, Op2	Compare	N,Z,C,V	4.4.6 CMP and CMN on page 108
CPSID	i	Change Processor State, Disable Interrupts	-	4.13.3 CPS on page 208
CPSIE	i	Change Processor State, Enable Interrupts	-	4.13.3 CPS on page 208
DMB	{opt}	Data Memory Barrier	-	4.13.5 DMB on page 209
DSB	{opt}	Data Synchronization Barrier	-	4.13.6 DSB on page 210
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
FLDMDBX ,FLDMIAX	Rn	FLDMX (Decrement Before, Increment After) loads	-	4.12.2 FLDMDBX, FLDMIAX on page 180
FSTMDBX,FSTMIAX	Rn	FSTMX (Decrement Before, Increment After) stores	-	4.12.3 FSTMDBX, FSTMIAX on page 181
ISB	{opt}	Instruction Synchronization Barrier	-	4.13.7 ISB on page 210
IT	-	If Then condition block	-	4.11.5 IT on page 175
LDA	Rd, [Rn]	Load-Acquire Word		4.14.10 LDA and STL on page 230
LDAB	Rd, [Rn]	Load-Acquire Byte		4.14.10 LDA and STL on page 230
LDAEX	Rd, [Rn]	Load-Acquire Exclusive Word	-	4.14.12 LDAEX and STLEX on page 233
LDAEXB	Rd, [Rn]	Load-Acquire Exclusive Byte	-	4.14.12 LDAEX and STLEX on page 233
LDAEXH	Rd, [Rn]	Load-Acquire Exclusive Halfword	-	4.14.12 LDAEX and STLEX on page 233
LDAH	Rd, [Rn]	Load-Acquire Halfword	-	4.14.10 LDA and STL on page 230
LDM	Rn{!}, reglist	Load Multiple	-	4.14.7 LDM and STM on page 227
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple Decrement Before	-	4.14.7 LDM and STM on page 227
LDMIA, LDMFD	Rn{!}, reglist	Load Multiple, Increment After	-	4.14.7 LDM and STM on page 227
LDR	Rt, [Rn, Rm {, LSL #shift}]	Load Register Word (register offset)	-	4.14.4 LDR and STR, register offset on page 223
LDR	Rt, label	Load Register Word (literal)	-	4.14.6 LDR, PC-relative on page 225

Mnemonic	Operands	Brief description	Flags	Page
LDR, LDRT	Rt, [Rn, #offset]	Load Register Word (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
LDRB	Rt, [Rn, Rm {, LSL #shift}]	Load Register Byte (register offset)	-	4.14.4 LDR and STR, register offset on page 223
LDRB	Rt, label	Load Register Byte (literal)	-	4.14.6 LDR, PC-relative on page 225
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register Byte (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
LDRD	Rt, Rt2, [Rn, #offset]	Load Register Dual (immediate offset)	-	4.14.3 LDR and STR, immediate offset on page 220
LDRD	Rt, Rt2, label	Load Register Dual (PC- relative)	-	4.14.6 LDR, PC-relative on page 225
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	-	4.14.11 LDREX and STREX on page 232
LDREXB	Rt, [Rn]	Load Register Exclusive Byte	-	4.14.11 LDREX and STREX on page 232
LDREXH	Rt, [Rn]	Load Register Exclusive Halfword	-	4.14.11 LDREX and STREX on page 232
LDRH	Rt, [Rn, Rm {, LSL #shift}]	Load Register Halfword (register offset)	-	4.14.4 LDR and STR, register offset on page 223
LDRH	Rt, label	Load Register Halfword (literal)	-	4.14.6 LDR, PC-relative on page 225
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register Halfword (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
LDRSB	Rt, [Rn, Rm {, LSL #shift}]	Load Register Signed Byte (register offset)	-	4.14.4 LDR and STR, register offset on page 223
LDRSB	Rt, label	Load Register Signed Byte (PC-relative)	-	4.14.6 LDR, PC-relative on page 225
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register Signed Byte (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
LDRSH	Rt, [Rn, Rm {, LSL #shift}]	Load Register Signed Halfword (register offset)	-	4.14.4 LDR and STR, register offset on page 223
LDRSH	Rt, label	Load Register Signed Halfword (PC-relative)	-	4.14.6 LDR, PC-relative on page 225
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register Signed Halfword (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
MCR,MCR2	{*cond*} *coproc*, #*opc1*, *Rt*, *CRn*, *CRm*{, #*opc2*}	Move to Coprocessor from Register	-	4.5.4 MCR and MCR2 on page 132
MCRR,MCRR2	{*cond*} *coproc*, #*opc1*, *Rt*, *Rt2*, *CRm*	Move to Coprocessor from two Registers	-	4.5.5 MCRR and MCRR2 on page 133

Mnemonic	Operands	Brief description	Flags	Page
MLA	Rd, Rn, Rm, Ra	Multiply Accumulate	-	4.7.2 MUL, MLA, and MLS on page 143
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract	-	4.7.2 MUL, MLA, and MLS on page 143
MOV, MOVS	Rd, Op2	Move	N,Z,C	4.4.7 MOV and MVN on page 109
MOV, MOVS	Rd, Rm	Move (register)	N,Z	4.4.7 MOV and MVN on page 109
MOVT	Rd, #imm16	Move Top	-	4.4.8 MOVT on page 111
MOVW	Rd, #imm16	Move 16-bit constant	N,Z,C	4.4.7 MOV and MVN on page 109
MRC,MRC2	{*cond*} *coproc*, *#opc1*, *Rt*, *CRn*, *CRm*{, *#opc2*}	Move to Register from |Coprocessor	-	4.5.6 MRC and MRC2 on page 133
MRRC,MRRC2	{*cond*} *coproc*, *#opc1*, *Rt*, *Rt2*, *CRm*	Move to two Registers from Coprocessor.	-	4.5.7 MRRC and MRRC2 on page 134
MRS	Rd, spec_reg	Move from Special Register to general register	-	4.13.8 MRS on page 211
MSR	spec_reg, Rn	Move from general register to Special Register	-	4.13.9 MSR on page 212
MUL, MULS	{Rd,} Rn, Rm	Multiply	N,Z	4.7.2 MUL, MLA, and MLS on page 143
MVN, MVNS	Rd, Op2	Bitwise NOT	N,Z,C	4.4.7 MOV and MVN on page 109
NOP	-	No Operation	-	4.13.10 NOP on page 213
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
PKHTB, PKHBT	{Rd,} Rn, Rm, {, Op2}	Pack Halfword	-	4.9.2 PKHBT and PKHTB on page 166
PLD	[Rn {, #offset}]	Preload Data	-	4.14.8 PLD on page 229
POP	reglist	Pop registers from stack	-	4.14.9 PUSH and POP on page 229
PUSH	reglist	Push registers onto stack	-	4.14.9 PUSH and POP on page 229
QADD	{Rd,} Rn, Rm	Saturating Add	Q	4.8.4 QADD and QSUB on page 160
QADD16	{Rd,} Rn, Rm	Saturating Add 16	-	4.8.4 QADD and QSUB on page 160
QADD8	{Rd,} Rn, Rm	Saturating Add 8	-	4.8.4 QADD and QSUB on page 160
QASX	{Rd,} Rn, Rm	Saturating Add and Subtract with Exchange	-	4.8.5 QASX and QSAX on page 161
QDADD	{Rd,} Rn, Rm	Saturating Double and Add	Q	4.8.6 QDADD and QDSUB on page 162
QDSUB	{Rd,} Rn, Rm	Saturating Double and Subtract	Q	4.8.6 QDADD and QDSUB on page 162
QSAX	{Rd,} Rn, Rm	Saturating Subtract and Add with Exchange	-	4.8.5 QASX and QSAX on page 161
QSUB	{Rd,} Rn, Rm	Saturating Subtract	Q	4.8.4 QADD and QSUB on page 160
QSUB16	{Rd,} Rn, Rm	Saturating Subtract 16	-	4.8.4 QADD and QSUB on page 160
QSUB8	{Rd,} Rn, Rm	Saturating Subtract 8	-	4.8.4 QADD and QSUB on page 160
RBIT	Rd, Rn	Reverse Bits	-	4.4.9 REV, REV16, REVSH, and RBIT on page 112

Mnemonic	Operands	Brief description	Flags	Page
REV	Rd, Rn	Reverse byte order in a word	-	4.4.9 REV, REV16, REVSH, and RBIT on page 112
REV16	Rd, Rn	Reverse byte order in each halfword	-	4.4.9 REV, REV16, REVSH, and RBIT on page 112
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-	4.4.9 REV, REV16, REVSH, and RBIT on page 112
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SADD16	{Rd,} Rn, Rm	Signed Add 16	GE	4.4.10 SADD16 and SADD8 on page 113
SADD8	{Rd,} Rn, Rm	Signed Add 8	GE	4.4.10 SADD16 and SADD8 on page 113
SASX	{Rd,} Rn, Rm	Signed Add and Subtract with Exchange	GE	4.4.11 SASX and SSAX on page 114
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	-	4.10.3 SBFX and UBFX on page 171
SDIV	{Rd,} Rn, Rm	Signed Divide	-	4.7.3 SDIV and UDIV on page 144
SEL	{Rd,} Rn, Rm	Select bytes	GE	4.4.12 SEL on page 116
SEV	-	Send Event	-	4.13.11 SEV on page 213
SG	-	Secure Gateway	-	4.13.12 SG on page 214
SHADD16	{Rd,} Rn, Rm	Signed Halving Add 16	-	4.4.13 SHADD16 and SHADD8 on page 116
SHADD8	{Rd,} Rn, Rm	Signed Halving Add 8	-	4.4.13 SHADD16 and SHADD8 on page 116
SHASX	{Rd,} Rn, Rm	Signed Halving Add and Subtract with Exchange	-	4.4.14 SHASX and SHSAX on page 117
SHSAX	{Rd,} Rn, Rm	Signed Halving Subtract and Add with Exchange	-	4.4.14 SHASX and SHSAX on page 117
SHSUB16	{Rd,} Rn, Rm	Signed Halving Subtract 16	-	4.4.15 SHSUB16 and SHSUB8 on page 118
SHSUB8	{Rd,} Rn, Rm	Signed Halving Subtract 8	-	4.4.15 SHSUB16 and SHSUB8 on page 118
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate halfwords	Q	4.7.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT on page 145
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Dual	Q	4.7.5 SMLAD and SMLADX on page 146
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long (32 × 32 + 64), 64-bit result	-	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155

Mnemonic	Operands	Brief description	Flags	Page
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long, halfwords	-	4.7.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT on page 147
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long Dual	-	4.7.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT on page 147
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate, word by halfword	Q	4.7.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT on page 145
SMLSD, SMLSDX	Rd, Rn, Rm, Ra	Signed Multiply Subtract Dual	Q	4.7.7 SMLSD and SMLSLD on page 149
SMLSLD, SMLSLDX	RdLo, RdHi, Rn, Rm	Signed Multiply Subtract Long Dual	-	4.7.7 SMLSD and SMLSLD on page 149
SMMLA, SMMLAR	Rd, Rn, Rm, Ra	Signed Most Significant Word Multiply Accumulate	-	4.7.8 SMMLA and SMMLS on page 151
SMMLS, SMMLSR	Rd, Rn, Rm, Ra	Signed Most Significant Word Multiply Subtract	-	4.7.8 SMMLA and SMMLS on page 151
SMMUL, SMMULR	Rd, Rn, Rm	Signed Most Significant Word Multiply	-	4.7.9 SMMUL on page 152
SMUAD, SMUADX	{Rd,} Rn, Rm	Signed Dual Multiply Add	Q.	4.7.10 SMUAD and SMUSD on page 153
SMULBB, SMULBT, SMULTB, SMULTT	{Rd,} Rn, Rm	Signed Multiply (halfwords)	-	4.7.11 SMUL and SMULW on page 154
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply Long (32 × 32), 64-bit result	-	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
SMULWB, SMULWT	{Rd,} Rn, Rm	Signed Multiply word by halfword	-	4.7.11 SMUL and SMULW on page 154
SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed Dual Multiply Subtract	-	4.7.10 SMUAD and SMUSD on page 153
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q	4.8.2 SSAT and USAT on page 158
SSAT16	Rd, #n, Rm	Signed Saturate 16	Q	4.8.3 SSAT16 and USAT16 on page 159
SSAX	{Rd,} Rn, Rm	Signed Subtract and Add with Exchange	GE	4.4.11 SASX and SSAX on page 114
SSUB16	{Rd,} Rn, Rm	Signed Subtract 16	GE	4.4.16 SSUB16 and SSUB8 on page 119
SSUB8	{Rd,} Rn, Rm	Signed Subtract 8	GE	4.4.16 SSUB16 and SSUB8 on page 119
STL	Rt, [Rn]	Store-Release Word	-	4.14.10 LDA and STL on page 230
STLB	Rt, [Rn]	Store-Release Byte	-	4.14.10 LDA and STL on page 230
STLEX	Rt, Rt [Rn]	Store-Release Exclusive Word	-	4.14.12 LDAEX and STLEX on page 233
STLEXB	Rt, Rt [Rn]	Store-Release Exclusive Byte	-	4.14.12 LDAEX and STLEX on page 233

Mnemonic	Operands	Brief description	Flags	Page
STLEXH	Rt, Rt [Rn]	Store-Release Exclusive Halfword	-	4.14.12 LDAEX and STLEX on page 233
STLH	Rt, [Rn]	Store-Release Halfword	-	4.14.10 LDA and STL on page 230
STM	Rn{!}, reglist	Store Multiple	-	4.14.7 LDM and STM on page 227
STMDB, STMEA	Rn{!}, reglist	Store Multiple Decrement Before	-	4.14.7 LDM and STM on page 227
STMIA, STMFD	Rn{!}, reglist	Store Multiple Increment After	-	4.14.7 LDM and STM on page 227
STR	Rt, [Rn, Rm {, LSL #shift}]	Store Register Word (register offset)	-	4.14.4 LDR and STR, register offset on page 223
STR, STRT	Rt, [Rn, #offset]	Store Register Word (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
STRB	Rt, [Rn, Rm {, LSL #shift}]	Store Register Byte (register offset)	-	4.14.4 LDR and STR, register offset on page 223
STRB, STRBT	Rt, [Rn, #offset]	Store Register Byte (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
STRD	Rt, Rt2, [Rn, #offset]	Store Register Dual two words	-	4.14.3 LDR and STR, immediate offset on page 220
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	-	4.14.11 LDREX and STREX on page 232
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	-	4.14.11 LDREX and STREX on page 232
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword	-	4.14.11 LDREX and STREX on page 232
STRH	Rt, [Rn, Rm {, LSL #shift}]	Store Register Halfword (register offset)	-	4.14.4 LDR and STR, register offset on page 223
STRH, STRHT	Rt, [Rn, #offset]	Store Register Halfword (immediate offset, unprivileged)	-	4.14.3 LDR and STR, immediate offset on page 220, 4.14.5 LDR and STR, unprivileged on page 224
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	-	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SVC	#imm	Supervisor Call	-	4.13.13 SVC on page 214
SXTAB	{Rd,} Rn, Rm {,ROR #n}	Sign extend 8 bits to 32 and Add	-	4.9.3 SXTA and UXTA on page 167
SXTAB16	{Rd,} Rn, Rm {,ROR #n}	Sign extend two 8-bit values to 16 and Add	-	4.9.3 SXTA and UXTA on page 167
SXTAH	{Rd,} Rn, Rm {,ROR #n}	Sign extend 16 bits to 32 and Add	-	4.9.3 SXTA and UXTA on page 167
SXTB	Rd, Rm {,ROR #n}	Sign extend 8 bits to 32	-	4.9.4 SXT and UXT on page 169
SXTB16	{Rd,} Rm {,ROR #n}	Sign extend 8 bits to 16	-	4.9.4 SXT and UXT on page 169
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a Halfword to 32	-	4.9.4 SXT and UXT on page 169

Mnemonic	Operands	Brief description	Flags	Page
TBB	[Rn, Rm]	Table Branch Byte	-	4.11.6 TBB and TBH on page 177
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword	-	4.11.6 TBB and TBH on page 177
TEQ	Rn, Op2	Test Equivalence	N,Z,C	4.4.17 TST and TEQ on page 121
TST	Rn, Op2	Test	N,Z,C	4.4.17 TST and TEQ on page 121
TT	Rd, [Rn]	Test Target	-	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTA	Rd, [Rn]	Test Target Alternate Domain	-	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTAT	Rd, [Rn]	Test Target Alternate Domain Unprivileged	-	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTT	Rd, [Rn]	Test Target Unprivileged	-	4.13.14 TT, TTT, TTA, and TTAT on page 215
UADD16	{Rd,} Rn, Rm	Unsigned Add 16	GE	4.4.18 UADD16 and UADD8 on page 122
UADD8	{Rd,} Rn, Rm	Unsigned Add 8	GE	4.4.18 UADD16 and UADD8 on page 122
UASX	{Rd,} Rn, Rm	Unsigned Add and Subtract with Exchange	GE	4.4.19 UASX and USAX on page 123
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	-	4.10.3 SBFX and UBFX on page 171
UDF	{c}{q} {#}imm	Permanently Undefined.	-	4.13.15 Cortex-M33 UGRM: UDF on page 217
UDIV	{Rd,} Rn, Rm	Unsigned Divide	-	4.7.3 SDIV and UDIV on page 144
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16	-	4.4.20 UHADD16 and UHADD8 on page 125
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8	-	4.4.20 UHADD16 and UHADD8 on page 125
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange	-	4.4.21 UHASX and UHSAX on page 126
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange	-	4.4.21 UHASX and UHSAX on page 126
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16	-	4.4.22 UHSUB16 and UHSUB8 on page 127
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8	-	4.4.22 UHSUB16 and UHSUB8 on page 127
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long (32 × 32 + 32 + 32), 64-bit result	-	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Long (32 × 32 + 64), 64-bit result	-	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Long (32 × 32), 64-bit result	-	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	-	4.8.8 UQADD and UQSUB on page 164
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	-	4.8.8 UQADD and UQSUB on page 164

Mnemonic	Operands	Brief description	Flags	Page
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	-	4.8.7 UQASX and UQSAX on page 163
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	-	4.8.7 UQASX and UQSAX on page 163
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	-	4.8.8 UQADD and UQSUB on page 164
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	-	4.8.8 UQADD and UQSUB on page 164
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	-	4.4.23 USAD8 on page 128
USADA8	Rd, Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	-	4.4.24 USADA8 on page 129
USAT	Rd, #n, Rm{,shift #s}, Ra	Unsigned Saturate	Q	4.8.2 SSAT and USAT on page 158
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q	4.8.3 SSAT16 and USAT16 on page 159
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE	4.4.19 UASX and USAX on page 123
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE	4.4.25 USUB16 and USUB8 on page 129
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE	4.4.25 USUB16 and USUB8 on page 129
UXTAB	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend 8 bits to 32 and Add	-	4.9.3 SXTA and UXTA on page 167
UXTAB16	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend two 8-bit values to 16 and Add	-	4.9.3 SXTA and UXTA on page 167
UXTAH	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend and Add Halfword	-	4.9.3 SXTA and UXTA on page 167
UXTB	Rd, Rm {,ROR #n}	Unsigned zero-extend Byte	-	4.9.4 SXT and UXT on page 169
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned zero-extend Byte 16	-	4.9.4 SXT and UXT on page 169
UXTH	Rd, Rm {,ROR #n}	Unsigned zero-extend Halfword	-	4.9.4 SXT and UXT on page 169
VABS	.F32 Sd, Sm	Floating-point Absolute	-	4.12.4 VABS on page 182
VADD	.F32 {Sd,} Sn, Sm	Floating-point Add	-	4.12.5 VADD on page 182
VCMP	.F32 Sd, <<Sm|#0.0>	Compare two floating-point registers, or one floating-point register and zero	N,Z,C,V	4.12.6 VCMP and VCMPE on page 183
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	4.12.6 VCMP and VCMPE on page 183
VCVT	.F32.Tm <Sd>, Sm	Convert from floating-point to integer	-	4.12.7 VCVT and VCVTR between floating-point and integer on page 184
VCVT	.Td.F32 Sd, Sd, #fbits	Convert from floating-point to fixed point	-	4.12.8 VCVT between floating-point and fixed-point on page 184

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	-	4.12.36 VCVTA, VCVTM VCVTN, and VCVTP on page 203
VCVTB VCVTT	.F32.F16 Sd, Sm	Convert half-precision value to single-precision or double-precision	-	4.12.37 VCVTB and VCVTT on page 204
VCVTB VCVTT	.F16.F32 Sd, Sm	Convert single-precision or double-precision register to half-precision	-	4.12.37 VCVTB and VCVTT on page 204
VCVTM	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding towards Minus infinity	-	4.12.36 VCVTA, VCVTM VCVTN, and VCVTP on page 203
VCVTN	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding to nearest with Ties to even	-	4.12.36 VCVTA, VCVTM VCVTN, and VCVTP on page 203
VCVTP	.Tm.F32 <Sd>, Sm	Convert from floating-point to integer with directed rounding towards Plus infinity	-	4.12.36 VCVTA, VCVTM VCVTN, and VCVTP on page 203
VCVTR	.Tm.F32 <Sd>, Sm	Convert between floating-point and integer with rounding.	-	4.12.7 VCVT and VCVTR between floating-point and integer on page 184
VDIV	.F32 {Sd,} Sn, Sm	Floating-point Divide	-	4.12.9 VDIV on page 186
VFMA	.F32 {Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate	-	4.12.10 VFMA and VFMS on page 186
VFMS	.F32 {Sd,} Sn, Sm	Floating-point Fused Multiply Subtract	-	4.12.10 VFMA and VFMS on page 186
VFNMA	.F32 {Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate	-	4.12.11 VFNMA and VFNMS on page 187
VFNMS	.F32 {Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract	-	4.12.11 VFNMA and VFNMS on page 187
VLDM	{mode}{.size} Rn{!}, list	Floating-point Load Multiple extension registers	-	4.12.12 VLDM on page 188
VLDR	.F32 Sd, [<Rn> {, #offset}]	Floating-point Load an extension register from memory (immediate)	-	4.12.13 VLDR on page 189
VLDR	.F32 Sd, <label>	Load an extension register from memory	-	4.12.13 VLDR on page 189
VLDR	.F32 Sd, [PC,#-0]	Load an extension register from memory	-	4.12.13 VLDR on page 189
VLLDM	<c> Rn	Floating-point Lazy Load multiple	-	4.12.14 VLLDM on page 189
VLSTM	<c> Rn	Floating-point Lazy Store multiple	-	4.12.15 VLSTM on page 190
VMAXNM	.F32 Sd, Sn, Sm	Maximum of two floating-point numbers with IEEE754-2008 NaN handling	-	4.12.38 VMAXNM and VMINNM on page 204
VMINNM	.F32 Sd, Sn, Sm	Minimum of two floating-point numbers with IEEE754-2008 NaN handling	-	4.12.38 VMAXNM and VMINNM on page 204
VMLA	.F32 Sd, Sn, Sm	Floating-point Multiply Accumulate	-	4.12.16 VMLA and VMLS on page 191
VMLS	.F32 Sd, Sn, Sm	Floating-point Multiply Subtract	-	4.12.16 VMLA and VMLS on page 191
VMOV	<Sn|Rt>, <Rt|Sn>	Copy core register to single-precision	-	4.12.20 VMOV core register to single-precision on page 193

Mnemonic	Operands	Brief description	Flags	Page
VMOV	<Sm|Rt>, <Sm1|Rt2>, <Rt|Sm>, <Rt2|Sm1>	Copy two core registers to two single-precision	-	4.12.21 VMOV two core registers to two single-precision registers on page 193
VMOV	{.size} Dd[x], Rt	Copy core register to scalar	-	4.12.23 VMOV core register to scalar on page 195
VMOV	{.dt} Rt, Dn[x]	Copy scalar to core register	-	4.12.19 VMOV scalar to core register on page 192
VMOV	.F32 Sd, #imm	Floating-point Move immediate	-	4.12.17 VMOV Immediate on page 191
VMOV	.F32 Sd, Sd, Sm	Copies the contents of one register to another	-	4.12.18 VMOV Register on page 192
VMOV	<Dm|Rt>, <Rt|Rt2>, <Rt2|Dm>	Floating-point Move transfers two words between two core registers and a doubleword register	-	4.12.22 VMOV two core registers and a double-precision register on page 194
VMRS	Rt, FPSCR	Move to core register from floating-point Special Register	N,Z,C,V	4.12.24 VMRS on page 195
VMSR	FPSCR, Rt	Move to floating-point Special Register from core register	-	4.12.25 VMSR on page 196
VMUL	.F32 {Sd,} Sn, Sm	Floating-point Multiply	-	4.12.26 VMUL on page 197
VNEG	.F32 Sd, Sm	Floating-point Negate	-	4.12.27 VNEG on page 197
VNMLA	.F32 Sd, Sn, Sm	Floating-point Multiply Accumulate and Negate	-	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VNMLS	.F32 Sd, Sn, Sm	Floating-point Multiply, Subtract and Negate	-	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VNMUL	.F32 {Sd,} Sn, Sm	Floating-point Multiply and Negate	-	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VPOP	{.size} list	Load multiple consecutive floating-point registers from the stack	-	4.12.29 VPOP on page 198
VPUSH	{.size} list	Store multiple consecutive floating-point registers to the stack	-	4.12.30 VPUSH on page 199
VRINTA	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties Away	-	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VRINTM	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Minus infinity	-	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VRINTN	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties to even	-	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VRINTP	.F32 Sd, Sm	Float to integer in floating-point format conversion with directed rounding to Plus infinity	-	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VRINTR	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding towards value specified in FPSCR	-	4.12.39 VRINTR and VRINTX on page 205
VRINTX	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding specified in FPSCR	-	4.12.39 VRINTR and VRINTX on page 205

Mnemonic	Operands	Brief description	Flags	Page
VRINTZ	.F32 Sd, Sm	Float to integer in floating-point format conversion with rounding towards Zero	-	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VSEL	.F32 Sd, Sn, Sm	Select register, alternative to a pair of conditional VMOV	-	4.12.35 VSEL on page 202
VSQRT	.F32 Sd, Sm	Calculates floating-point Square Root	-	4.12.31 VSQRT on page 200
VSTM	{mode}{.size} Rn{!}, list	Floating-point Store Multiple	-	4.12.32 VSTM on page 200
VSTR	.F32 Sd, [Rn{, #offset}]	Floating-point Store Register stores an extension register to memory	-	4.12.33 VSTR on page 201
VSUB	F32 {Sd,} Sn, Sm	Floating-point Subtract	-	4.12.34 VSUB on page 202
WFE	-	Wait For Event	-	4.13.16 WFE on page 217
WFI	-	Wait For Interrupt	-	4.13.17 WFI on page 218
YIELD	-	Suspend task	-	4.13.18 YIELD on page 218

4.1.1 Binary compatibility with other Cortex processors

The processor implements the T32 instruction set and features provided by the Arm®v8-M architecture profile. There are restrictions on moving code designed for processors that are implementations of the Arm®v6-M or Arm®v7-M architectures.

If code designed for other Cortex®-M processors relies on memory protection, it cannot be moved to the Cortex®-M33 processor. In this case, the memory protection scheme and driver code must be updated from PMSAv7 to PMSAv8.

If Cortex®-M33 is configured without floating-point, any Arm®v7-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.

4.2 CMSIS functions

ISO/IEC C code cannot directly access some Cortex®-M33 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.

4.2.1 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 4-2: CMSIS functions to generate some Cortex®-M33 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)
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)

Instruction	CMSIS function
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)

4.2.2 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.

4.2.3 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 4-3: 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)
xPSR	Read	uint32_t __get_xPSR (void)
BASEPRI_MAX	Write	void __set_BASEPRI_MAX (uint32_t basePri)
FPSCR	Read	uint32_t __get_FPSCR (void)

Special register	Access	CMSIS function
	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)`

4.2.4 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 4-4: 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)`

4.3 About the instruction descriptions

Additional information about using the instructions, including operands, restrictions when using PC or SP, flexible second operand, and shift operations.

4.3.1 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.

4.3.2 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.

- In an implementation with Arm®v8-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®-M33 processor only supports T32 instructions.

4.3.3 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.

4.3.3.1 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.

4.3.3.1.1 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`.

4.3.3.2 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, $1 \le n \le 32$.

LSL #*n*

Logical shift left *n* bits, $1 \le n \le 31$.

LSR #*n*

Logical shift right *n* bits, $1 \le n \le 32$.

ROR #*n*

Rotate right *n* bits, $1 \le n \le 31$.

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.

4.3.4 Shift Operations

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

Register shift can be performed:

- Directly by the instructions `ASR`, `LSR`, `LSL`, `ROR`, and `RRX`, 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 or the Flexible second operand. 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.

4.3.4.1 ASR

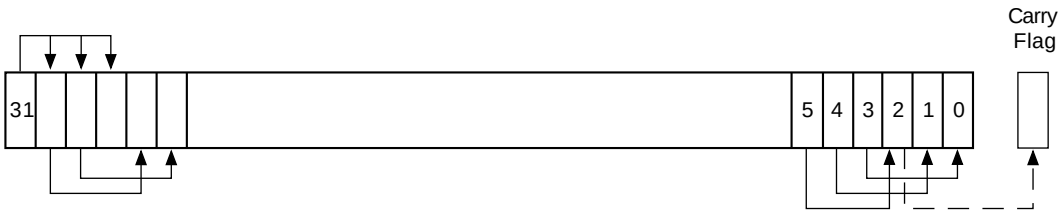
Arithmetic shift right by *n* bits moves the left-hand `32-`*n* bits of the register *Rm*, to the right by *n* places, into the right-hand `32-`*n* bits of the result. And it copies the original bit[31] of the register into the left-hand *n* bits of the result.

You can use the ASR #*n* operation to divide the value in the register *Rm* by 2^n, with the result being rounded towards negative-infinity.

When the instruction is `ASRS` or when ASR #*n* is used in *Operand2* with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[*n*-1], of the register *Rm*.

- If *n* is 32 or more, then all the bits in the result are set to the value of bit[31] of *Rm*.
- If *n* is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of *Rm*.

Figure 4-1: ASR #3



4.3.4.2 LSR

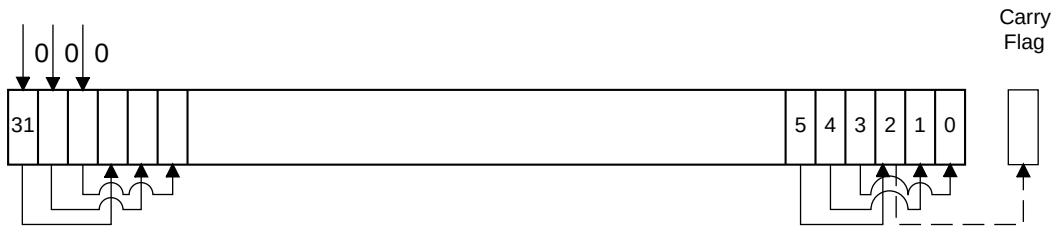
Logical shift right by `n` bits moves the left-hand `32-n` bits of the register `Rm`, to the right by `n` places, into the right-hand `32-n` bits of the result. And it sets the left-hand `n` bits of the result to 0.

You can use the LSR `#n` operation to divide the value in the register `Rm` by 2^n, if the value is regarded as an unsigned integer.

When the instruction is `LSRS` or when LSR `#n` is used in `Operand2` with the instructions `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` or `TST`, the carry flag is updated to the last bit shifted out, bit[`n`-1], of the register `Rm`.

- If `n` is 32 or more, then all the bits in the result are cleared to 0.
- If `n` is 33 or more and the carry flag is updated, it is updated to 0.

Figure 4-2: LSR #3



4.3.4.3 LSL

Logical shift left by *n* bits moves the right-hand 32-*n* bits of the register *Rm*, to the left by *n* places, into the left-hand 32-*n* bits of the result. And it sets the right-hand *n* bits of the result to 0.

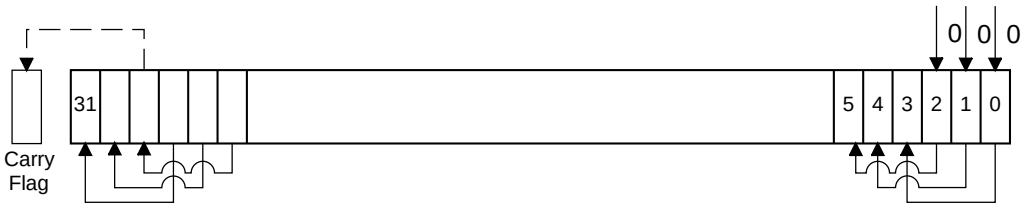
You can use the LSL #*n* operation to multiply the value in the register *Rm* by 2^n, if the value is regarded as an unsigned integer or a two's complement signed integer. Overflow can occur without warning.

When the instruction is LSLS or when LSL #*n*, with non-zero *n*, is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit shifted out, bit[32-*n*], of the register *Rm*. These instructions do not affect the carry flag when used with LSL #0.

Note

- If *n* is 32 or more, then all the bits in the result are cleared to 0.
- If *n* is 33 or more and the carry flag is updated, it is updated to 0.

Figure 4-3: LSL #3



4.3.4.4 ROR

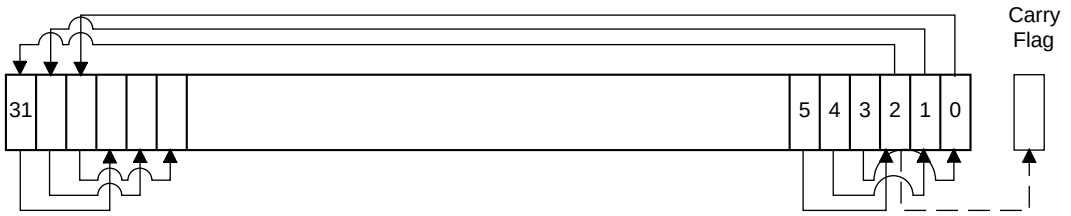
Rotate right by *n* bits moves the left-hand 32-*n* bits of the register *Rm*, to the right by *n* places, into the right-hand 32-*n* bits of the result. And it moves the right-hand *n* bits of the register into the left-hand *n* bits of the result.

When the instruction is RORS or when ROR #*n* is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to the last bit rotation, bit[*n*-1], of the register *Rm*.

Note

- If *n* is 32, then the value of the result is same as the value in *Rm*, and if the carry flag is updated, it is updated to bit[31] of *Rm*.
- ROR with shift length, *n*, more than 32 is the same as ROR with shift length *n*-32.

Figure 4-4: ROR #3

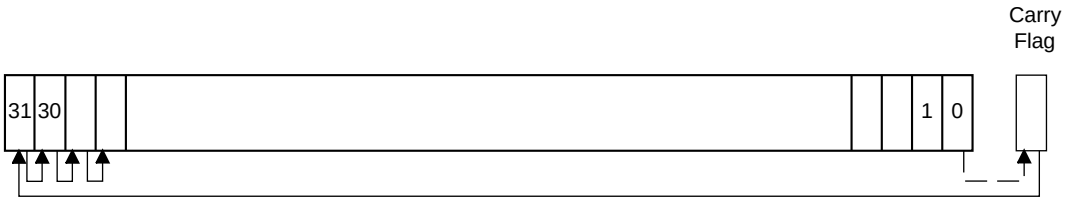


4.3.4.5 RRX

Rotate right with extend moves the bits of the register *Rm* to the right by one bit. And it copies the carry flag into bit[31] of the result.

When the instruction is RRXS or when RRX is used in *Operand2* with the instructions MOVS, MVNS, ANDS, ORRS, ORNS, EORS, BICS, TEQ or TST, the carry flag is updated to bit[0] of the register *Rm*.

Figure 4-5: RRX



4.3.5 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®-M33 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.

4.3.6 PC-relative 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.

- 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]`.

4.3.7 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.

4.3.7.1 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.

For more information about APSR, see 3.1.3.6.1 Application Program Status Register on page 30

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.

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

4.3.7.2 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 4-5: 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.
`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.

Suffix	Flags	Meaning
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.
```

4.3.8 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.
```

4.4 General data processing instructions

Reference material for the Cortex®-M33 processor data processing instruction set.

4.4.1 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 4-6: Data processing instructions

Mnemonic	Brief description	See
ADC	Add with Carry	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
ADD	Add	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
ADDW	Add	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
AND	Logical AND	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
ASR	Arithmetic Shift Right	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
BIC	Bit Clear	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
CLZ	Count leading zeros	4.4.5 CLZ on page 108
CMN	Compare Negative	4.4.6 CMP and CMN on page 108
CMP	Compare	4.4.6 CMP and CMN on page 108
EOR	Exclusive OR	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
LSL	Logical Shift Left	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
LSR	Logical Shift Right	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
MOV	Move	4.4.7 MOV and MVN on page 109
MOVT	Move Top	4.4.8 MOVT on page 111
MOVW	Move 16-bit constant	4.4.7 MOV and MVN on page 109
MVN	Move NOT	4.4.7 MOV and MVN on page 109
ORN	Logical OR NOT	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
ORR	Logical OR	4.4.3 AND, ORR, EOR, BIC, and ORN on page 105
RBIT	Reverse Bits	4.4.9 REV, REV16, REVSH, and RBIT on page 112
REV	Reverse byte order in a word	4.4.9 REV, REV16, REVSH, and RBIT on page 112
REV16	Reverse byte order in each halfword	4.4.9 REV, REV16, REVSH, and RBIT on page 112
REVSH	Reverse byte order in bottom halfword and sign extend	4.4.9 REV, REV16, REVSH, and RBIT on page 112
ROR	Rotate Right	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
RRX	Rotate Right with Extend	4.4.4 ASR, LSL, LSR, ROR, and RRX on page 107
RSB	Reverse Subtract	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SADD16	Signed Add 16	4.4.10 SADD16 and SADD8 on page 113
SADD8	Signed Add 8	4.4.10 SADD16 and SADD8 on page 113
SASX	Signed Add and Subtract with Exchange	4.4.11 SASX and SSAX on page 114
SEL	Select bytes	4.4.12 SEL on page 116

Mnemonic	Brief description	See
SSAX	Signed Subtract and Add with Exchange	4.4.11 SASX and SSAX on page 114
SBC	Subtract with Carry	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SHADD16	Signed Halving Add 16	4.4.13 SHADD16 and SHADD8 on page 116
SHADD8	Signed Halving Add 8	4.4.13 SHADD16 and SHADD8 on page 116
SHASX	Signed Halving Add and Subtract with Exchange	4.4.14 SHASX and SHSAX on page 117
SHSAX	Signed Halving Subtract and Add with Exchange	4.4.14 SHASX and SHSAX on page 117
SHSUB16	Signed Halving Subtract 16	4.4.15 SHSUB16 and SHSUB8 on page 118
SHSUB8	Signed Halving Subtract 8	4.4.15 SHSUB16 and SHSUB8 on page 118
SSUB16	Signed Subtract 16	4.4.16 SSUB16 and SSUB8 on page 119
SSUB8	Signed Subtract 8	4.4.16 SSUB16 and SSUB8 on page 119
SUB	Subtract	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
SUBW	Subtract	4.4.2 ADD, ADC, SUB, SBC, and RSB on page 103
TEQ	Test Equivalence	4.4.17 TST and TEQ on page 121
TST	Test	4.4.17 TST and TEQ on page 121
UADD16	Unsigned Add 16	4.4.18 UADD16 and UADD8 on page 122
UADD8	Unsigned Add 8	4.4.18 UADD16 and UADD8 on page 122
UASX	Unsigned Add and Subtract with Exchange	4.4.19 UASX and USAX on page 123
USAX	Unsigned Subtract and Add with Exchange	4.4.19 UASX and USAX on page 123
UHADD16	Unsigned Halving Add 16	4.4.20 UHADD16 and UHADD8 on page 125
UHADD8	Unsigned Halving Add 8	4.4.20 UHADD16 and UHADD8 on page 125
UHASX	Unsigned Halving Add and Subtract with Exchange	4.4.21 UHASX and UHSAX on page 126
UHSAX	Unsigned Halving Subtract and Add with Exchange	4.4.21 UHASX and UHSAX on page 126
UHSUB16	Unsigned Halving Subtract 16	4.4.22 UHSUB16 and UHSUB8 on page 127
UHSUB8	Unsigned Halving Subtract 8	4.4.22 UHSUB16 and UHSUB8 on page 127
USAD8	Unsigned Sum of Absolute Differences	4.4.23 USAD8 on page 128
USADA8	Unsigned Sum of Absolute Differences and Accumulate	4.4.24 USADA8 on page 129
USUB16	Unsigned Subtract 16	4.4.25 USUB16 and USUB8 on page 129
USUB8	Unsigned Subtract 8	4.4.25 USUB16 and USUB8 on page 129

4.4.2 ADD, ADC, SUB, SBC, and RSB

Add, Add with carry, Subtract, Subtract with carry, and Reverse Subtract.

```
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.
- *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.

```
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.
```

4.4.3 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

`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.

```
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
```

4.4.4 ASR, LSL, LSR, ROR, and RRX

Arithmetic Shift Right, Logical Shift Left, Logical Shift Right, Rotate Right, and Rotate Right with Extend.

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.

```
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.
```

4.4.5 CLZ

Count Leading Zeros.

`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.

```
CLZ     R4,R9
CLZNE   R2,R3
```

4.4.6 CMP and CMN

Compare and Compare Negative.

`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.

```
CMP     R2, R9
CMN     R0, #6400
CMPGT   SP, R7, LSL #2
```

4.4.7 MOV and MVN

Move and Move NOT.

```
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.


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.

```
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.
```

4.4.8 MOVT

Move Top.

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.

```
MOVT   R3, #0xF123 ; Write 0xF123 to upper halfword of R3, lower halfword
```

```
                    ; and APSR are unchanged.
```

4.4.9 REV, REV16, REVSH, and RBIT

Reverse bytes and Reverse bits.

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.

```
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.
```

4.4.10 SADD16 and SADD8

Signed Add 16 and Signed Add 8.

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:

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.

The SADD8 instruction:

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';
```

```
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.
```

4.4.11 SASX and SSAX

Signed Add and Subtract with Exchange and Signed Subtract and Add with Exchange.

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 top halfword of the second operand from the bottom 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 subtraction to the top halfword of the destination register.
3. Adds the signed bottom halfword of the first operand with the signed top halfword of the second operand.
4. Writes the signed 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 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';
```

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';
```

```
SASX    R0, R4, R5     ; Adds top halfword of R4 to bottom halfword of R5 and
                       ; writes to top halfword of R0.
                       ; Subtracts top halfword of R5 from bottom halfword of R4
                       ; and writes to bottom halfword of R0.
SSAX    R7, R3, R2     ; Subtracts bottom halfword of R2 from top halfword of R3
                       ; and writes to top halfword of R7.
                       ; Adds bottom halfword of R3 with top halfword of R2 and
```

```
                        ; writes to bottom halfword of R7.
```

4.4.12 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.

`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.

```
SADD16 R0, R1, R2     ; Set GE bits based on result.
SEL R0, R0, R3        ; Select bytes from R0 or R3, based on GE.
```

4.4.13 SHADD16 and SHADD8

Signed Halving Add 16 and Signed Halving Add 8.

`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.

```
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.
```

4.4.14 SHASX and SHSAX

Signed Halving Add and Subtract with Exchange and Signed Halving Subtract and Add with Exchange.

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.

```
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.
```

4.4.15 SHSUB16 and SHSUB8

Signed Halving Subtract 16 and Signed Halving Subtract 8.

```
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: The SHSUBB8 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.

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.

```
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.
```

4.4.16 SSUB16 and SSUB8

Signed Subtract 16 and Signed Subtract 8.

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>;
   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';
```

```
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.
```

4.4.17 TST and TEQ

Test bits and Test Equivalence.

`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.

```
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.
```

4.4.18 UADD16 and UADD8

Unsigned Add 16 and Unsigned Add 8.

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';
   APSR.GE<2> = if sum3 >= 0x100 then '1' else '0';
   APSR.GE<3> = if sum4 >= 0x100 then '1' else '0';
```

```
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.
```

4.4.19 UASX and USAX

Unsigned Add and Subtract with Exchange and Unsigned Subtract and Add with Exchange.

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';
```

```
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.
```

4.4.20 UHADD16 and UHADD8

Unsigned Halving Add 16 and Unsigned Halving Add 8.

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.

```
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.
```

4.4.21 UHASX and UHSAX

Unsigned Halving Add and Subtract with Exchange and Unsigned Halving Subtract and Add with Exchange.

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.

```
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.
                         ; Adds top halfword of R5 to bottom halfword of R3 and
                         ; writes halved result to bottom halfword of R0.
```

4.4.22 UHSUB16 and UHSUB8

Unsigned Halving Subtract 16 and Unsigned Halving Subtract 8.

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.

```
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.
```

4.4.23 USAD8

Unsigned Sum of Absolute Differences.

`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.

```
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.
```

4.4.24 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

`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.

```
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.
```

4.4.25 USUB16 and USUB8

Unsigned Subtract 16 and Unsigned Subtract 8.

`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';
    APSR.GE<2> = if diff3 >= 0 then '1' else '0';
    APSR.GE<3> = if diff4 >= 0 then '1' else '0';
```

```
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.
```

4.5 Coprocessor instructions

Reference material for the Cortex®-M33 processor coprocessor instruction set.

4.5.1 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 4-7: Coprocessor instructions

Mnemonic	Brief description	See
CDP, CDP2	Coprocessor data processing	4.5.3 CDP and CDP2 on page 132
MCR, MCR2	Move to Coprocessor from Register	4.5.4 MCR and MCR2 on page 132
MCRR, MCRR2	Move to Coprocessor from two Registers	4.5.5 MCRR and MCRR2 on page 133
MRC, MRC2	Move to Register from Coprocessor	4.5.6 MRC and MRC2 on page 133
MRRC, MRRC2	Move to two Registers from Coprocessor	4.5.7 MRRC and MRRC2 on page 134

4.5.2 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`

Intrinsics	Equivalent Instruction
`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`

4.5.3 CDP and CDP2

Coprocessor Data Processing tells a coprocessor to perform an operation.

`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.

4.5.4 MCR and MCR2

Move to Coprocessor from Register. Depending on the coprocessor, you might be able to specify various additional operations.

`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.

4.5.5 MCRR and MCRR2

Move to Coprocessor from two Registers. Depending on the coprocessor, you might be able to specify various additional operations.

`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.

4.5.6 MRC and MRC2

Move to Register from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

`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.

4.5.7 MRRC and MRRC2

Move to two Registers from Coprocessor. Depending on the coprocessor, you might be able to specify various additional operations.

```
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.

4.6 CDE instructions

Reference material for the Cortex®-M33 processor *Custom Datapath Extension* (CDE) instruction set for the implementation of *Arm Custom Instructions* (ACIs).

4.6.1 List of CDE instructions

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

Table 4-10: Coprocessor instructions

Mnemonic	Brief description	See
`CX1{A}`	Custom Instruction Class 1	4.6.2 CX1{A} on page 135
`CX1D{A}`	Custom Instruction Class 1 Dual	4.6.3 CX1D{A} on page 135
`CX2{A}`	Custom Instruction Class 2	4.6.4 CX2{A} on page 136
`CX2D{A}`	Custom Instruction Class 2 Dual	4.6.5 CX2D{A} on page 137
`CX3{A}`	Custom Instruction Class 3	4.6.6 CX3{A} on page 137

Mnemonic	Brief description	See
CX3D{A}	Custom Instruction Class 3 Dual	4.6.7 CX3D{A} on page 138
VCX1{A}	Custom Extension Instruction Class 1	4.6.8 VCX1{A} on page 139
VCX2{A}	Custom Extension Instruction Class 2	4.6.9 VCX2{A} on page 140
VCX3{A}	Custom Extension Instruction Class 3	4.6.10 VCX3{A} on page 141

4.6.2 CX1{A}

Custom Instruction Class 1 {Accumulation}. Custom Instruction Class 1 computes a value based on an immediate, and optionally the destination value, and writes the result to the destination register.

The source and destination registers can be either general-purpose registers or the condition flags, specified by use of APSR_nzcv.

```
CX1  <coproc>, <Rd>, #<imm>

CX1A{cond}  <coproc>, <Rd>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter is either:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "*Rd*" field. For accumulator variants <Rd> also specifies the source register. APSR_nzcv is encoded by the "*Rd*" field value 0b1111.

<imm> is the immediate encoded in op1:op2:op3.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.3 CX1D{A}

Custom Instruction Class 1 Dual {Accumulation}. Custom Instruction Class 1 Dual computes a value based on an immediate, and optionally the destination register pair value, and writes the result to a destination register pair.

The destination registers are a consecutive pair of general-purpose registers. The significance of the words in each pair is consistent with the current data endianness.

```
CX1D  <coproc>, <Rd>, <Rd+1>, #<imm>
```

```
CX1DA{cond} <coproc>, <Rd>, <Rd+1>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter is either:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose R0 - R10 specifying the first of destination register pair, encoded in the "*Rd*" field. For accumulator variants *<Rd>* also specifies the source register.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.4 CX2{A}

Custom Instruction Class 2 {Accumulation}. Custom Instruction Class 2 computes a value based on a source register, an immediate, and optionally the destination value, and writes the result to the destination register.

The source and destination registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

```
CX2 <coproc>, <Rd>, <Rn>, #<imm>

CX2A{cond} <coproc>, <Rd>, <Rn>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "*Rd*" field. For accumulator variants *<Rd>* also specifies the source register. APSR_nzcv is encoded by the "*Rd*" field value `0b1111`.

<Rn> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rn*" field. APSR_nzcv is encoded by the "*Rn*" field value `0b1111`.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.5 CX2D{A}

Custom Instruction Class 2 Dual {Accumulation}. Custom instruction Class 2 Dual computes a value based on a source register, an immediate, and optionally the destination register pair value, and writes the result to the destination register pair.

The destination registers are a consecutive pair of general-purpose registers. The source registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv. The significance of the words in each pair is consistent with the current data endianness.

```
CX2D <coproc>, <Rd>, <Rd+1>, <Rn>, #<imm>

CX2DA{cond} <coproc>, <Rd>, <Rd+1>, <Rn>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose R0 - R10 specifying the first of destination register pair, encoded in the "*Rd*" field. For accumulator variants *<Rd>* also specifies the source register.

<Rn> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rn*" field. APSR_nzcv is encoded by the "*Rn*" field value `0b1111`.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.6 CX3{A}

Custom Instruction Class 3 {Accumulation}. Custom Instruction Class 3 computes a value based on two source registers, an immediate and optionally the destination value, and writes the result to the destination register.

The source and destination registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

```
CX3 <coproc>, <Rd>, <Rn, <Rm>, #<imm>

CX3A{cond} <coproc>, <Rd>, <Rn, <Rm>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "*Rd*" field. For accumulator variants *<Rd>* also specifies the source register. APSR_nzcv is encoded by the "*Rd*" field value `0b1111`.

<Rn> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rn*" field. APSR_nzcv is encoded by the "*Rn*" field value `0b1111`.

<Rm> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rm*" field. APSR_nzcv is encoded by the "*Rm*" field value `0b1111`.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.7 CX3D{A}

Custom Instruction Class 3 Dual {Accumulation}. Custom Instruction Class 3 Dual computes a value baed on two source registers, an immediate and optionally the destination register pair value, and writes the result to the destination register pair.

The source registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv. The destination registers are a consecutive pair of general-purpose registers. The significance of the words in each pair is consistent with the current data endianness.

```
CX3D <coproc>, <Rd>, <Rd+1>, <Rn, <Rm>, #<imm>

CX3DA{cond} <coproc>, <Rd>, <Rd+1>, <Rn, <Rm>, #<imm>
```

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Rd> is the general-purpose register R0 - R10 specifying the first of destination register pair,encoded in the "*Rd*" field. For accumulator variants, *<Rd>* also specifies the source register.

<Rn> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rn*" field. APSR_nzcv is encoded by the "*Rn*" field value `0b1111`.

<Rm> is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "*Rm*" field. APSR_nzcv is encoded by the "*Rm*" field value `0b1111`.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.8 VCX1{A}

Custom Extension Instruction Class 1 {Accumulation}. Custom Extension instruction class 1 computes a value based on an immediate and optionally the destination value, and writes the result to the destination register.

The source and destination registers are within the floating-point register file, and require the current execution state to have access to these registers.

`VCX1 <coproc>, <Dd>, #<imm>` (Double-register non-accumulator variant)

`VCX1A <coproc>, <Dd>, #<imm>` (Double-register accumulator variant)

`VCX1 <coproc>, <Sd>, #<imm>` (Single-register non-accumulator variant)

`VCX1A <coproc>, <Sd>, #<imm>` (Single-register accumulator variant)

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Dd> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "D:Vd" fields.

<Sd> is the 32-bit name of the floating-point source and destination register S0 - S31 encoded in the "Vd:D" fields.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.9 VCX2{A}

Custom Extension Instruction Class 2 {Accumulation}. Custom Extension instruction class 2 computes a value based on a source register, an immediate and optionally the destination value, and writes the result to the destination register.

The source and destination registers are within the floating-point register file, and require the current execution state to have access to these registers.

`VCX2 <coproc>, <Dd>, <Dm>, #<imm>` (Double-register non-accumulator variant)

`VCX2A <coproc>, <Dd>, <Dm>, #<imm>` (Double-register accumulator variant)

`VCX2 <coproc>, <Sd>, <Sm>, #<imm>` (Single-register non-accumulator variant)

`VCX2A <coproc>, <Sd>, <Sm>, #<imm>` (Single-register accumulator variant)

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Dd> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "D:Vd" fields.

<Dm> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "M:Vm" fields.

<Sd> is the 32-bit name of the floating-point source and destination register S0 - S31 encoded in the "Vd:D" fields.

<Sm> is the 32-bit name of the floating-point source register S0 - S31 encoded in the "Vm:M" fields.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.6.10 VCX3{A}

Custom Extension Instruction Class 3 {Accumulation}. Custom Extension instruction class 3 computes a value based on two source registers, an immediate and optionally the destination value, and writes the result to the destination register.

The source and destination registers are within the floating-point register file, and require the current execution state to have access to these registers.

`VCX3 <coproc>, <Dd>, <Dn>, <Dm>, #<imm>` (Double-register non-accumulator variant)

`VCX3A <coproc>, <Dd>, <Dn>, <Dm>, #<imm>` (Double-register accumulator variant)

`VCX3 <coproc>, <Sd>, <Sn>, <Sm>, #<imm>` (Single-register non-accumulator variant)

`VCX3A <coproc>, <Sd>, <Sn>, <Sm>, #<imm>` (Single-register accumulator variant)

Where:

A is for Accumulate with existing register contents. This parameter must be one of the following values:

Absent	Encoded as A = 0
Present	Encoded as A =1

<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.

<Dd> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "D:Vd" fields.

<Dm> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "M:Vm" fields.

<Dn> is the 64-bit name of the floating-point source and destination register D0 - D15 encoded in the "N:Vn" fields.

<Sd> is the 32-bit name of the floating-point source register S0 - S31 encoded in the "Vd:D" fields.

<Sm> is the 32-bit name of the floating-point source register S0 - S31 encoded in the "Vm:M" fields.

<Sn> is the 32-bit name of the floating-point source and destination register S0 - S31 encoded in the "Vn:N" fields.

<imm> is the immediate encoded in `op1:op2:op3`.

Operation

The operation of these instructions can be customized depending on the coprocessor number.

4.7 Multiply and divide instructions

Reference material for the Cortex®-M33 processor multiply and divide instruction set.

4.7.1 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 4-11: Multiply and divide instructions

Mnemonic	Brief description	See
`MLA`	Multiply with Accumulate, 32-bit result	4.7.2 MUL, MLA, and MLS on page 143
`MLS`	Multiply and Subtract, 32-bit result	4.7.2 MUL, MLA, and MLS on page 143
`MUL`	Multiply, 32-bit result	4.7.2 MUL, MLA, and MLS on page 143
`SDIV`	Signed Divide	4.7.3 SDIV and UDIV on page 144
`SMLA[B,T]`	Signed Multiply Accumulate (halfwords)	4.7.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT on page 145
`SMLAD`, `SMLADX`	Signed Multiply Accumulate Dual	4.7.5 SMLAD and SMLADX on page 146
`SMLAL`	Signed Multiply with Accumulate (32 × 32 + 64), 64-bit result	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
`SMLAL[B,T]`	Signed Multiply Accumulate Long (halfwords)	4.7.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT on page 147
`SMLALD`, `SMLALDX`	Signed Multiply Accumulate Long Dual	4.7.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT on page 147
`SMLAW[B|T]`	Signed Multiply Accumulate (word by halfword)	4.7.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT on page 145
`SMLSD`	Signed Multiply Subtract Dual	4.7.7 SMLSD and SMLSLD on page 149
`SMLSLD`	Signed Multiply Subtract Long Dual	4.7.7 SMLSD and SMLSLD on page 149
`SMMLA`	Signed Most Significant Word Multiply Accumulate	4.7.8 SMMLA and SMMLS on page 151
`SMMLS`, `SMMLSR`	Signed Most Significant Word Multiply Subtract	4.7.8 SMMLA and SMMLS on page 151
`SMMUL`, `SMMULR`	Signed Most Significant Word Multiply	4.7.9 SMMUL on page 152
`SMUAD`, `SMUADX`	Signed Dual Multiply Add	4.7.10 SMUAD and SMUSD on page 153
`SMUL[B,T]`	Signed Multiply (word by halfword)	4.7.11 SMUL and SMULW on page 154
`SMULL`	Signed Multiply (32 × 32), 64-bit result	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
`SMULWB`, `SMULWT`	Signed Multiply (word by halfword)	4.7.11 SMUL and SMULW on page 154
`SMUSDX` ,`SMUSD`	Signed Dual Multiply Subtract	4.7.10 SMUAD and SMUSD on page 153
`UDIV`	Unsigned Divide	4.7.3 SDIV and UDIV on page 144

Mnemonic	Brief description	See
UMAAL	Unsigned Multiply Accumulate Accumulate Long (32 × 32 + 32 + 32), 64-bit result	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
UMLAL	Unsigned Multiply with Accumulate (32 × 32 + 64), 64-bit result	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155
UMULL	Unsigned Multiply (32 × 32), 64-bit result	4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL on page 155

4.7.2 MUL, MLA, and MLS

Multiply, Multiply with Accumulate, and Multiply with Subtract, using 32-bit operands, and producing a 32-bit result.

```
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.

```
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)
```

4.7.3 SDIV and UDIV

Signed Divide and Unsigned Divide.

`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®-M33 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.

```
SDIV  R0, R2, R4  ; Signed divide, R0 = R2/R4
```

```
UDIV  R8, R8, R1  ; Unsigned divide, R8 = R8/R1
```

4.7.4 SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT

Signed Multiply Accumulate (halfwords).

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.

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.

```
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.
```

4.7.5 SMLAD and SMLADX

Signed Multiply Accumulate Long Dual, Signed Multiply Accumulate Long Dual exchange.

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.

```
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.
```

4.7.6 SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT

Signed Multiply Accumulate Long Dual and Signed Multiply Accumulate Long (halfwords).

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:

- 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.

```
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.
```

4.7.7 SMLSD and SMLSLD

Signed Multiply Subtract Dual and Signed Multiply Subtract Long Dual.

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*.

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.

```
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.
```

4.7.8 SMMLA and SMMLS

Signed Most Significant Word Multiply Accumulate and Signed Most Significant Word Multiply Subtract.

`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.

```
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
                            ; 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.
```

4.7.9 SMMUL

Signed Most Significant Word Multiply.

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.

```
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.
```

4.7.10 SMUAD and SMUSD

Signed Dual Multiply Add and Signed Dual Multiply Subtract.

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.

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.

```
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.
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.
```

4.7.11 SMUL and SMULW

Signed Multiply (halfwords) and Signed Multiply (word by halfword).

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.

```
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
                              ; 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.
```

4.7.12 UMULL, UMAAL, UMLAL, SMULL, and SMLAL

Signed and Unsigned Multiply Long, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

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.

```
UMULL       R0, R4, R5, R6   ; Unsigned (R4,R0) = R5 × R6
SMLAL       R4, R5, R3, R8   ; Signed (R5,R4) = (R5,R4) + R3 × R8
```

4.8 Saturating instructions

Reference material for the Cortex®-M33 processor saturating instruction set.

4.8.1 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 4-12: Saturating instructions

Mnemonic	Brief description	See
QADD	Saturating Add	4.8.4 QADD and QSUB on page 160
QASX	Saturating Add and Subtract with Exchange	4.8.5 QASX and QSAX on page 161
QDADD	Saturating Double and Add	4.8.6 QDADD and QDSUB on page 162
QDSUB	Saturating Double and Subtract	4.8.6 QDADD and QDSUB on page 162
QSAX	Saturating Subtract and Add with Exchange	4.8.5 QASX and QSAX on page 161
QSUB	Saturating Subtract	4.8.4 QADD and QSUB on page 160
QSUB16	Saturating Subtract 16	4.8.4 QADD and QSUB on page 160
SSAT	Signed Saturate	4.8.2 SSAT and USAT on page 158
SSAT16	Signed Saturate Halfword	4.8.3 SSAT16 and USAT16 on page 159
UQADD16	Unsigned Saturating Add 16	4.8.8 UQADD and UQSUB on page 164
UQADD8	Unsigned Saturating Add 8	4.8.8 UQADD and UQSUB on page 164
UQASX	Unsigned Saturating Add and Subtract with Exchange	4.8.7 UQASX and UQSAX on page 163
UQSAX	Unsigned Saturating Subtract and Add with Exchange	4.8.7 UQASX and UQSAX on page 163
UQSUB16	Unsigned Saturating Subtract 16	4.8.8 UQADD and UQSUB on page 164
UQSUB8	Unsigned Saturating Subtract 8	4.8.8 UQADD and UQSUB on page 164
USAT	Unsigned Saturate	4.8.2 SSAT and USAT on page 158
USAT16	Unsigned Saturate Halfword	4.8.3 SSAT16 and USAT16 on page 159

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.

4.8.2 SSAT and USAT

Signed Saturate and Unsigned Saturate to any bit position, with optional shift before saturating.

`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 $-2^{n-1} \le x \le 2^{n-1}-1$.

The USAT instruction applies the specified shift, then saturates to the unsigned range $0 \le x \le 2^{n}-1$.

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.

```
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.
```

4.8.3 SSAT16 and USAT16

Signed Saturate and Unsigned Saturate to any bit position for two halfwords.

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.

```
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.
```

4.8.4 QADD and QSUB

Saturating Add and Saturating Subtract, signed.

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 $-2^{n-1} \le x \le 2^{n-1}-1$, 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.

```
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
                     ; 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.
```

4.8.5 QASX and QSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, signed.

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 $-215 \le x \le 215 - 1$, 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 $-215 \le x \le 215 - 1$, 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 $-215 \le x \le 215 - 1$, 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 $-215 \le x \le 215 - 1$, 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.

```
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.
```

4.8.6 QDADD and QDSUB

Saturating Double and Add and Saturating Double and Subtract, signed.

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 ≤ *x* ≤ 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.

```
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.
```

4.8.7 UQASX and UQSAX

Saturating Add and Subtract with Exchange and Saturating Subtract and Add with Exchange, unsigned.

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 ≤ *x* ≤ 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 ≤ *x* ≤ 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 $0 \le x \le 216 - 1$, 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 $0 \le x \le 216 - 1$, 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.

```
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.
```

4.8.8 UQADD and UQSUB

Saturating Add and Saturating Subtract Unsigned.

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 $0 \le x \le 2^{16}-1$, 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 $0 \le x \le 2^{8}-1$, 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 $0 \le x \le 2^{16}-1$, 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 $0 \le x \le 2^{8}-1$, 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.

```
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.
```

4.9 Packing and unpacking instructions

Reference material for the Cortex®-M33 processor packing and unpacking instruction set.

4.9.1 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 4-13: Packing and unpacking instructions

Mnemonic	Brief description	See
PKH	Pack Halfword	4.9.2 PKHBT and PKHTB on page 166
SXTAB	Extend 8 bits to 32 and add	4.9.3 SXTA and UXTA on page 167
SXTAB16	Dual extend 8 bits to 16 and add	4.9.3 SXTA and UXTA on page 167
SXTAH	Extend 16 bits to 32 and add	4.9.3 SXTA and UXTA on page 167
SXTB	Sign extend a byte	4.9.4 SXT and UXT on page 169
SXTB16	Dual extend 8 bits to 16 and add	4.9.4 SXT and UXT on page 169
SXTH	Sign extend a halfword	4.9.4 SXT and UXT on page 169
UXTAB	Extend 8 bits to 32 and add	4.9.3 SXTA and UXTA on page 167
UXTAB16	Dual extend 8 bits to 16 and add	4.9.3 SXTA and UXTA on page 167
UXTAH	Extend 16 bits to 32 and add	4.9.3 SXTA and UXTA on page 167
UXTB	Zero extend a byte	4.9.4 SXT and UXT on page 169
UXTB16	Dual zero extend 8 bits to 16 and add	4.9.4 SXT and UXT on page 169
UXTH	Zero extend a halfword	4.9.4 SXT and UXT on page 169

4.9.2 PKHBT and PKHTB

Pack Halfword.

```
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.

```
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.
```

4.9.3 SXTA and UXTA

Signed and Unsigned Extend and Add.

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.

Condition flags

These instructions do not affect the flags.

```
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.
```

4.9.4 SXT and UXT

Sign extend and Zero extend.

`SXT`*op*`{`*cond*`}` *Rd*`,` *Rn* `{,` ROR `#`*n*`}`

`UXT`*op*`{`*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.

```
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.
```

4.10 Bit field instructions

Reference material for the Cortex®-M33 processor bit field instruction set.

4.10.1 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 4-14: Bit field instructions

Mnemonic	Brief description	See
`BFC`	Bit Field Clear	4.10.2 BFC and BFI on page 170
`BFI`	Bit Field Insert	4.10.2 BFC and BFI on page 170
`SBFX`	Signed Bit Field Extract	4.10.3 SBFX and UBFX on page 171
`UBFX`	Unsigned Bit Field Extract	4.10.3 SBFX and UBFX on page 171

4.10.2 BFC and BFI

Bit Field Clear and Bit Field Insert.

`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.

```
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.
```

4.10.3 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

```
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.

```
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.
```

4.11 Branch and control instructions

Reference material for the Cortex®-M33 processor branch and control instruction set.

4.11.1 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 4-15: Branch and control instructions

Mnemonic	Brief description	See
B	Branch	4.11.2 B, BL, BX, and BLX on page 172
BL	Branch with Link	4.11.2 B, BL, BX, and BLX on page 172
BLX	Branch indirect with Link	4.11.2 B, BL, BX, and BLX on page 172
BLXNS	Branch indirect with Link, Non-secure	4.11.3 BXNS and BLXNS on page 174
BX	Branch indirect	4.11.2 B, BL, BX, and BLX on page 172
BXNS	Branch indirect, Non-secure	4.11.3 BXNS and BLXNS on page 174
CBNZ	Compare and Branch if Non Zero	4.11.4 CBZ and CBNZ on page 174
CBZ	Compare and Branch if Zero	4.11.4 CBZ and CBNZ on page 174
IT	If-Then	4.11.5 IT on page 175
TBB	Table Branch Byte	4.11.6 TBB and TBH on page 177
TBH	Table Branch Halfword	4.11.6 TBB and TBH on page 177

4.11.2 B, BL, BX, and BLX

Branch instructions.

`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 4-16: Branch ranges

Instruction	Branch range
B *label*	–16MB to +16MB.
B*cond* *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

B*cond* 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.

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.
```

4.11.3 BXNS and BLXNS

Branch and Exchange Non-secure and Branch with Link and Exchange Non-secure.

`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

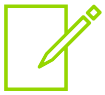
PC and SP cannot be used for *Rm*.

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*.

4.11.4 CBZ and CBNZ

Compare and Branch on Zero, Compare and Branch on Non-Zero.

`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.

```
CBZ    R5, target ; Forward branch if R5 is zero
CBNZ   R0, target ; Forward branch if R0 is not zero
```

4.11.5 IT

If-Then condition instruction.

`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.

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.

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.

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 `Bcond` must be inside an IT block. `Bcond` 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.

```
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
```

4.11.6 TBB and TBH

Table Branch Byte and Table Branch Halfword.

`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.

```
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
CaseC
; an instruction sequence follows
```

4.12 Floating-point instructions

Reference material for the Cortex®-M33 processor floating-point instruction set that the FPU uses.

4.12.1 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 4-17: Floating-point instructions

Mnemonic	Brief description	See
FLDMDBX	FLDMX (Decrement Before) loads multiple extension registers from consecutive memory locations	4.12.2 FLDMDBX, FLDMIAX on page 180
FLDMIAX	FLDMX (Increment After) loads multiple extension registers from consecutive memory locations	4.12.2 FLDMDBX, FLDMIAX on page 180
FSTMDBX	FSTMX (Decrement Before) stores multiple extension registers to consecutive memory locations	4.12.3 FSTMDBX, FSTMIAX on page 181
FSTMIAX	FSTMX (Increment After) stores multiple extension registers to consecutive memory locations	4.12.3 FSTMDBX, FSTMIAX on page 181
VABS	Floating-point Absolute	4.12.4 VABS on page 182
VADD	Floating-point Add	4.12.5 VADD on page 182
VCMP	Compare two floating-point registers, or one floating-point register and zero	4.12.6 VCMP and VCMPE on page 183
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	4.12.6 VCMP and VCMPE on page 183
VCVT	Convert between floating-point and integer	4.12.7 VCVT and VCVTR between floating-point and integer on page 184
VCVT	Convert between floating-point and fixed point	4.12.8 VCVT between floating-point and fixed-point on page 184
VCVTA, VCVTN, VCVTP, VCVTM	Float to integer conversion with directed rounding	4.12.36 VCVTA, VCVTM VCVTN, and VCVTP on page 203
VCVTB	Converts half-precision value to single-precision	4.12.37 VCVTB and VCVTT on page 204
VCVTR	Convert between floating-point and integer with rounding	4.12.7 VCVT and VCVTR between floating-point and integer on page 184
VCVTT	Converts single-precision register to half-precision	4.12.37 VCVTB and VCVTT on page 204
VDIV	Floating-point Divide	4.12.9 VDIV on page 186
VFMA	Floating-point Fused Multiply Accumulate	4.12.10 VFMA and VFMS on page 186
VFMS	Floating-point Fused Multiply Subtract	4.12.10 VFMA and VFMS on page 186
VFNMA	Floating-point Fused Negate Multiply Accumulate	4.12.11 VFNMA and VFNMS on page 187

Mnemonic	Brief description	See
VFNMS	Floating-point Fused Negate Multiply Subtract	4.12.11 VFNMA and VFNMS on page 187
VLDM	Load Multiple extension registers	4.12.12 VLDM on page 188
VLDR	Loads an extension register from memory	4.12.13 VLDR on page 189
VMAXNM, VMINNM	Maximum, Minimum with IEEE754-2008 NaN handling	4.12.38 VMAXNM and VMINNM on page 204
VMLA	Floating-point Multiply Accumulate	4.12.16 VMLA and VMLS on page 191
VMLS	Floating-point Multiply Subtract	4.12.16 VMLA and VMLS on page 191
VMOV	Floating-point Move Immediate	4.12.17 VMOV Immediate on page 191
VMOV	Floating-point Move Register	4.12.18 VMOV Register on page 192
VMOV	Copy Arm® core register to single-precision	4.12.20 VMOV core register to single-precision on page 193
VMOV	Copy 2 Arm® core registers to 2 single-precision	4.12.21 VMOV two core registers to two single-precision registers on page 193
VMOV	Copies between Arm® core register to scalar	4.12.23 VMOV core register to scalar on page 195
VMOV	Copies between Scalar to Arm® core register	4.12.19 VMOV scalar to core register on page 192
VMRS	Move to Arm® core register from floating-point System Register	4.12.24 VMRS on page 195
VMSR	Move to floating-point System Register from Arm® Core register	4.12.25 VMSR on page 196
VMUL	Multiply floating-point	4.12.26 VMUL on page 197
VNEG	Floating-point negate	4.12.27 VNEG on page 197
VNMLA	Floating-point multiply and add	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VNMLS	Floating-point multiply and subtract	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VNMUL	Floating-point multiply	4.12.28 VNMLA, VNMLS and VNMUL on page 198
VPOP	Pop extension registers	4.12.29 VPOP on page 198
VPUSH	Push extension registers	4.12.30 VPUSH on page 199
VRINTA, VRINTN, VRINTP, VRINTM	Float to integer (in floating-point format) conversion with directed rounding	4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ on page 206
VRINTR, VRINTX	Float to integer (in floating-point format) conversion	4.12.39 VRINTR and VRINTX on page 205
VSEL	Select register, alternative to a pair of conditional VMOV	4.12.35 VSEL on page 202
VSQRT	Floating-point square root	4.12.31 VSQRT on page 200
VSTM	Store Multiple extension registers	4.12.32 VSTM on page 200
VSTR	Stores an extension register to memory	4.12.33 VSTR on page 201
VSUB	Floating-point Subtract	4.12.34 VSUB on page 202

4.12.2 FLDMDBX, FLDMIAX

FLDMX (Decrement Before, Increment After) loads multiple extension registers from consecutive memory locations using an address from a general-purpose register.

```
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**.

4.12.3 FSTMDBX, FSTMIAX

FSTMX (Decrement Before, Increment After) stores multiple extension registers to consecutive memory locations using an address from a general-purpose register.

```
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**.

4.12.4 VABS

Floating-point Absolute.

```
VABS{cond}.F32 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.

```
VABS.F32 S4, S6
```

4.12.5 VADD

Floating-point Add.

```
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.

```
VADD.F32 S4, S6, S7
```

4.12.6 VCMP and VCMPE

Compares two floating-point registers, or one floating-point register and zero.

`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.

```
VCMP.F32    S4, #0.0VCMP.F32    S4, S2
```

4.12.7 VCVT and VCVTR between floating-point and integer

Converts a value in a register from floating-point to and from a 32-bit integer.

`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.

4.12.8 VCVT between floating-point and fixed-point

Converts a value in a register from floating-point to and from fixed-point.

```
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.

4.12.9 VDIV

Divides floating-point values.

```
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.

4.12.10 VFMA and VFMS

Floating-point Fused Multiply Accumulate and Subtract.

```
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.

4.12.11 VFNMA and VFNMS

Floating-point Fused Negate Multiply Accumulate and Subtract.

```
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.

4.12.12 VLDM

Floating-point Load Multiple.

```
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.

```
VLDMIA.F64 r1, {d3,d4,d5}
```

4.12.13 VLDR

Loads a single extension register from memory.

```
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.

4.12.14 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.

```
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.

4.12.15 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.

```
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.

4.12.16 VMLA and VMLS

Multiplies two floating-point values, and accumulates or subtracts the result.

```
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.

4.12.17 VMOV Immediate

Move floating-point Immediate.

```
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.

4.12.18 VMOV Register

Copies the contents of one register to another.

```
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.

4.12.19 VMOV scalar to core register

Transfers one word of a doubleword floating-point register to an Arm® core register.

```
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.

4.12.20 VMOV core register to single-precision

Transfers a single-precision register to and from an Arm® core register.

```
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.

4.12.21 VMOV two core registers to two single-precision registers

Transfers two consecutively numbered single-precision registers to and from two Arm® core registers.

```
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.

4.12.22 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.

```
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.

4.12.23 VMOV core register to scalar

Transfers one word to a floating-point register from an Arm® core register.

```
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.

4.12.24 VMRS

Move to Arm® Core register from floating-point System Register.

```
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.

4.12.25 VMSR

Move to floating-point System Register from Arm® Core register.

```
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.

4.12.26 VMUL

Floating-point Multiply.

```
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.

4.12.27 VNEG

Floating-point Negate.

```
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.

4.12.28 VNMLA, VNMLS and VNMUL

Floating-point multiply with negation followed by add or subtract.

```
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.

4.12.29 VPOP

Floating-point extension register Pop.

`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.

4.12.30 VPUSH

Floating-point extension register Push.

`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.

4.12.31 VSQRT

Floating-point Square Root.

```
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.

4.12.32 VSTM

Floating-point Store Multiple.

```
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.

4.12.33 VSTR

Floating-point Store.

```
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.

4.12.34 VSUB

Floating-point Subtract.

```
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.

4.12.35 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.

```
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.

4.12.36 VCVTA, VCVTM VCVTN, and VCVTP

Floating-point to integer conversion with directed rounding.

`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.

4.12.37 VCVTB and VCVTT

Converts between half-precision and single-precision without intermediate rounding.

`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.

4.12.38 VMAXNM and VMINNM

Return the minimum or the maximum of two floating-point numbers with NaN handling as specified by IEEE754-2008.

```
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.

4.12.39 VRINTR and VRINTX

Round a floating-point value to an integer in floating-point format.

```
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.

4.12.40 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ

Round a floating-point value to an integer in floating-point format using directed rounding.

```
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.

4.13 Miscellaneous instructions

Reference material for the Cortex®-M33 processor miscellaneous instructions.

4.13.1 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 4-18: Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	4.13.2 BKPT on page 207
CPSID	Change Processor State, Disable Interrupts	4.13.3 CPS on page 208
CPSIE	Change Processor State, Enable Interrupts	4.13.3 CPS on page 208
DMB	Data Memory Barrier	4.13.5 DMB on page 209
DSB	Data Synchronization Barrier	4.13.6 DSB on page 210
ISB	Instruction Synchronization Barrier	4.13.7 ISB on page 210
MRS	Move from special register to register	4.13.8 MRS on page 211
MSR	Move from register to special register	4.13.9 MSR on page 212
NOP	No Operation	4.13.10 NOP on page 213
SEV	Send Event	4.13.11 SEV on page 213
SG	Secure Gateway	4.13.12 SG on page 214
SVC	Supervisor Call	4.13.13 SVC on page 214
TT	Test Target	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTT	Test Target Unprivileged	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTA	Test Target Alternate Domain	4.13.14 TT, TTT, TTA, and TTAT on page 215
TTAT	Test Target Alternate Domain Unprivileged	4.13.14 TT, TTT, TTA, and TTAT on page 215
WFE	Wait For Event	4.13.16 WFE on page 217
WFI	Wait For Interrupt	4.13.17 WFI on page 218
YIELD	Yield	4.13.18 YIELD on page 218

4.13.2 BKPT

Breakpoint.

```
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.

```
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.

4.13.3 CPS

Change Processor State.

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.

```
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)
```

4.13.4 CPY

Copy is a pre-Unified Assembler Language (UAL) synonym for MOV (register).

`CPY` *`Rd`*`,` *`Rn`*

This is equivalent to:

`MOV` *`Rd`*`,` *`Rn`*

4.13.5 DMB

Data Memory Barrier.

`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.

```
DMB  ; Data Memory Barrier
```

4.13.6 DSB

Data Synchronization Barrier.

`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.

```
DSB ; Data Synchronisation Barrier
```

4.13.7 ISB

Instruction Synchronization Barrier.

`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.

```
ISB  ; Instruction Synchronisation Barrier
```

4.13.8 MRS

Move the contents of a special register to a general-purpose register.

```
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.

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.

```
MRS  R0, PRIMASK ; Read PRIMASK value and write it to R0
```

4.13.9 MSR

Move the contents of a general-purpose register into the specified special register.

`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.

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.

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*.

```
MSR  CONTROL, R1 ; Read R1 value and write it to the CONTROL register.
```

4.13.10 NOP

No Operation.

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.

```
NOP  ; No operation
```

4.13.11 SEV

Send Event.

`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.

```
SEV ; Send Event
```

4.13.12 SG

Secure Gateway.

`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*.

4.13.13 SVC

Supervisor Call.

`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.

```
SVC  #0x32     ; Supervisor Call (SVCall handler can extract the immediate value
               ; by locating it through the stacked PC)
```

4.13.14 TT, TTT, TTA, and TTAT

Test Target (Alternate Domain, Unprivileged).

`{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 4-19: 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.
[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 of the enabled MPU regions.
- The address matched multiple MPU regions.

- The TT instruction or TTT instruction was executed from unprivileged mode.

The TTA and TTAT instructions are UNDEFINED when executed from Non-secure state.

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.
- The TT instruction or TTT instruction was executed from unprivileged mode.

4.13.15 UDF

Permanently Undefined.

`UDF{cond}.W {#}imm`

Where:

imm Is a:

- 8-bit unsigned immediate, in the range 0 to 255. The PE ignores the value of this constant.
- 16-bit unsigned immediate, in the range 0 to 65535. The PE 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.

4.13.16 WFE

Wait For Event.

`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.

```
WFE  ; Wait for event
```

4.13.17 WFI

Wait for Interrupt.

`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.

```
WFI ; Wait for interrupt
```

4.13.18 YIELD

Yield

`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.

```
YIELD; Suspend task
```

4.14 Memory access instructions

Reference material for the Cortex®-M33 processor memory access instruction set.

4.14.1 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 4-20: Memory access instructions

Mnemonic	Brief description	See
`ADR`	Generate PC-relative address	4.14.2 ADR on page 220
`CLREX`	Clear Exclusive	4.14.13 CLREX on page 235
`LDM{mode}`	Load Multiple registers	4.14.7 LDM and STM on page 227
`LDA{type}`	Load-Acquire	4.14.10 LDA and STL on page 230
`LDAEX`	Load-Acquire Exclusive	4.14.12 LDAEX and STLEX on page 233
`LDR{type}`	Load Register using immediate offset	4.14.3 LDR and STR, immediate offset on page 220
`LDR{type}`	Load Register using register offset	4.14.4 LDR and STR, register offset on page 223
`LDR{type}T`	Load Register with unprivileged access	4.14.5 LDR and STR, unprivileged on page 224
`LDR`	Load Register using PC-relative address	4.14.6 LDR, PC-relative on page 225
`LDRD`	Load Register Dual	4.14.3 LDR and STR, immediate offset on page 220
`LDREX{type}`	Load Register Exclusive	4.14.11 LDREX and STREX on page 232
`PLD`	Preload Data.	4.14.8 PLD on page 229
`POP`	Pop registers from stack	4.14.9 PUSH and POP on page 229
`PUSH`	Push registers onto stack	4.14.9 PUSH and POP on page 229
`STL{mode}`	Store-Release	4.14.10 LDA and STL on page 230
`STLEX`	Store Release Exclusive	4.14.12 LDAEX and STLEX on page 233
`STM{mode}`	Store Multiple registers	4.14.7 LDM and STM on page 227
`STR{type}`	Store Register using immediate offset	4.14.3 LDR and STR, immediate offset on page 220
`STR{type}`	Store Register using register offset	4.14.4 LDR and STR, register offset on page 223

Mnemonic	Brief description	See
`STR{type}T`	Store Register with unprivileged access	4.14.5 LDR and STR, unprivileged on page 224
`STREX{type}`	Store Register Exclusive	4.14.11 LDREX and STREX on page 232

4.14.2 ADR

Generate PC-relative address.

`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.

```
ADR     R1, TextMessage   ; Write address value of a location labelled as
                          ; TextMessage to R1.
```

4.14.3 LDR and STR, immediate offset

Load and Store with immediate offset, pre-indexed immediate offset, or post-indexed immediate offset.

```
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]
```

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 4-21: 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.

```
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.
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.
```

4.14.4 LDR and STR, register offset

Load and Store with register offset.

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.

Condition flags

These instructions do not change the flags.

```
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.
```

4.14.5 LDR and STR, unprivileged

Load and Store with unprivileged access.

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.

```
STRBTEQ  R4, [R7]       ; Conditionally store least significant byte in
                        ; R4 to an address in R7, with unprivileged access.
LDRHT    R2, [R2, #8]   ; Load halfword value from an address equal to
                        ; sum of R2 and 8 into R2, with unprivileged access.
```

4.14.6 LDR, PC-relative

Load register from memory.

```
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 4-22: 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`*.

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.

```
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.
```

4.14.7 LDM and STM

Load and Store Multiple registers.

op{*addr_mode*}{*cond*} *Rn*{!}, *reglist*

Where:

op
Is one of:

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:

- *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.

```
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.
```

4.14.8 PLD

Preload Data.

```
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.

4.14.9 PUSH and POP

Push registers onto, and pop registers off a full-descending stack.

```
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.

```
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.
```

4.14.10 LDA and STL

Load-Acquire and Store-Release.

```
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.

4.14.11 LDREX and STREX

Load and Store Register Exclusive.

```
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.

```
    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.
```

4.14.12 LDAEX and STLEX

Load-Acquire and Store Release Exclusive.

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.

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.

```
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]
```

4.14.13 CLREX

Clear Exclusive.

`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.

```
CLREX
```

5. The Cortex®-M33 Peripherals

This chapter describes the Cortex®-M33 peripherals.

5.1 About the Cortex®-M33 peripherals

The address map of the *Private peripheral bus* (PPB).

Table 5-1: Core peripheral register regions

Address	Core peripheral	Description
0xE000E000-0xE000E00F	System control and ID registers	Includes the Interrupt Controller Type and Auxiliary Control registers
0xE000ED00-0xE000ED8F		5.2.1 System control block registers summary on page 237
0xE000EDF0-0xE000EEFF		Debug registers in the SCS
0xE000EF00-0xE000EF8F		Includes the SW Trigger Interrupt Register
0xE000E010-0xE000E0FF	System timer	5.3 System timer, SysTick on page 269
0xE000E100-0xE000ECFF	Nested Vectored Interrupt Controller registers	5.4 Nested Vectored Interrupt Controller on page 272
0xE000ED00-0xE000EDEF	Security Attribution Unit	5.5.1 Security Attribution Unit on page 283 -
0xE000ED90-0xE000EDB8	Memory Protection Unit	5.5.9 Memory Protection Unit on page 290
0xE000EF30-0xE000EF44	Floating-Point Unit	5.6 Floating-Point Unit on page 300

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.

[13] Software can read the MPU Type Register at 0xE000ED90 to test for the presence of a *Memory Protection Unit* (MPU).

- 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.

5.2 System Control Block

The *System Control Block* (SCB) provides system implementation information and system control that includes configuration, control, and reporting of system exceptions.

5.2.1 System control block registers summary

Reference information for the SCB registers.

Table 5-2: Summary of the system control block registers

Address	Name	Type	Required privilege	Reset value	Description
`0xE000E008`	ACTLR	RW	Privileged	`0x00000000`	5.2.2 Auxiliary Control Register - Cortex-M33 on page 238
`0xE000ED00`	CPUID	RO	Privileged	`0x411FD210`	5.2.3 CPUID Base Register - ARMv8M on page 239
`0xE000ED04`	ICSR	RW[14]	Privileged	`0x00000000`	5.2.4 M33 Interrupt Control and State Register on page 240
`0xE000ED08`	VTOR	RW	Privileged	UNKNOWN	5.2.5 Vector Table Offset Register on page 245
`0xE000ED0C`	AIRCR	RW[14]	Privileged	`0xFA050000`	5.2.6 Application Interrupt and Reset Control Register - ARMv8 on page 246
`0xE000ED10`	SCR	RW	Privileged	`0x00000000`	5.2.7 System Control Register - Cortex-M33 on page 249
`0xE000ED14`	CCR	RW	Privileged	`0x00000201`	5.2.8 Configuration and Control Register on page 251
`0xE000ED18`	SHPR1	RW	Privileged	`0x00000000`	5.2.9.1 System Handler Priority Register 1 -ARMv8M on page 254
`0xE000ED1C`	SHPR2	RW	Privileged	`0x00000000`	5.2.9.2 M33 System Handler Priority Register 2 on page 255
`0xE000ED20`	SHPR3	RW	Privileged	`0x00000000`	5.2.9.3 M33 System Handler Priority Register 3 on page 255
`0xE000ED24`	SHCSR	RW	Privileged	`0x00000000`	5.2.10 System Handler Control and State Register - ARMv8M on page 255
`0xE000ED28`	CFSR	RW	Privileged	`0x00000000`	5.2.11 M33 Configurable Fault Status Register on page 259
`0xE000ED28`	MMFSR[15]	RW	Privileged	`0x00`	5.2.11.1 M33 MemManage Fault Status Register on page 259
`0xE000ED29`	BFSR[15]	RW	Privileged	`0x00`	5.2.11.2 M33 BusFault Status Register on page 261
`0xE000ED2A`	UFSR[15]	RW	Privileged	`0x0000`	5.2.11.3 UsageFault Status Register on page 262
`0xE000ED2C`	HFSR	RW	Privileged	`0x00000000`	5.2.12 M33 HardFault Status Register on page 264

Address	Name	Type	Required privilege	Reset value	Description
0xE000ED34	MMFAR	RW	Privileged	UNKNOWN	5.2.13 M33 MemManage Fault Address Register on page 265
0xE000ED38	BFAR	RW	Privileged	UNKNOWN	5.2.14 BusFault Address Register on page 265
0xE000ED3C	AFSR	RAZ/WI	Privileged	-	Auxiliary Fault Status Register not implemented
0xE000ED88	CPACR	RW	Privileged	0x00000000	5.2.15 Coprocessor Access Control Register - ARMv8M on page 266
0xE000ED8C	NSACR	RW	Privileged	UNKNOWN	5.2.16 Non-secure Access Control Register on page 267

5.2.2 Auxiliary Control Register

The ACTLR provides disable bits for the FPU exception outputs, dual-issue functionality, flushing of the trace output from the ITM and DWT, Exclusive instruction control, out-of-order floating point instructions, and handling interruptible instructions.

By default, this register is set to provide optimum performance from the Cortex®-M33 processor and does not normally require modification.

See 5.2.1 System control block registers summary on page 237 for the ACTLR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The ACTLR bit assignments are:

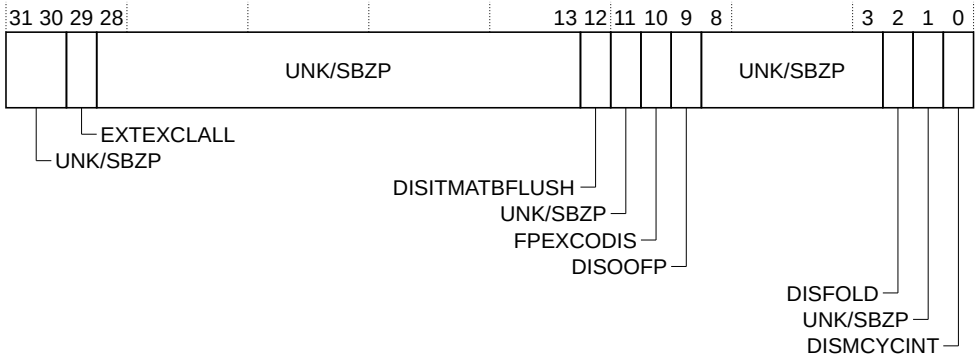


Table 5-3: ACTLR bit assignments

Bits	Name	Function
[31:30]	-	Reserved for software testing purposes only.

[14] See the register description for more information.
[15] A subregister of the CFSR.

Bits	Name	Function
[29]	EXTEXCLALL	**0** Normal operation. Memory requests on *Code region* AHB (C-AHB) or *System* AHB (S-AHB) interfaces associated with LDREX and STREX instructions only assert HEXCL and respond to HEXOKAY if the address is shareable. **1** All memory requests on C-AHB or S-AHB interfaces associated with LDREX and STREX instructions assert HEXCL and respond to HEXOKAY irrespective of the shareable attribute associated with the address. Setting EXTEXCLALL allows external exclusive operations to be used in a configuration with no MPU. This is because the default memory map does not include any shareable Normal memory.
[28:13]	-	Reserved. UNK/SBZP
[12]	DISITMATBFLUSH	Disables ITM and DWT ATB flush: **0** Normal operation. **1** ITM and DWT ATB flush disabled. AFVALID is ignored and AFREADY is held HIGH.
[11]	-	Reserved. UNK/SBZP
[10]	FPEXCODIS	Disables FPU exception outputs: **0** Normal operation. **1** FPU exception outputs are disabled.
[9]	DISOOFP	Disables floating-point instructions completing out of order with respect to the non-floating point instructions: **0** Normal operation. **1** Floating-point instructions completing out of order are disabled.
[8:3]	-	Reserved. UNK/SBZP
[2]	DISFOLD	Disables dual-issue functionality: **0** Normal operation. **1** Dual-issue functionality is disabled. Setting this bit reduces performance.
[1]	-	Reserved. UNK/SBZP
[0]	DISMCYCINT	Disables interruption of multi-cycle instructions: **0** Normal operation. **1** Disables interruption of multi-cycle instructions. This increases the interrupt latency of the processor because load, store, multiply, and divide operations complete before interrupt stacking occurs.

5.2.3 CPUID Base Register

The CPUID Base Register contains the processor part number, version, and implementation information.

See 5.2.1 System control block registers summary on page 237 for the CPUID attributes.

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

The bit assignments are:

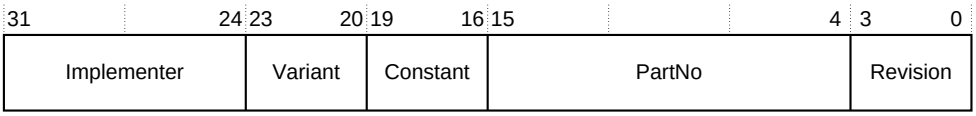


Table 5-4: CPUID bit assignments

Bits	Name	Function
[31:24]	Implementer	Implementer code: **0x41** Arm®
[23:20]	Variant	Variant number, the n value in the *rnpm* product revision identifier: **0x1** Revision 1
[19:16]	Constant	Reads as `0xF`
[15:4]	PartNo	Part number of the processor: **0xD21** Cortex®-M33
[3:0]	Revision	Revision number, the m value in the *rnpm* product revision identifier: **0x0** Patch 0.

5.2.4 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.

See 5.2.1 System control block registers summary on page 237 for the ICSR attributes.

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

The ICSR bit assignments are:

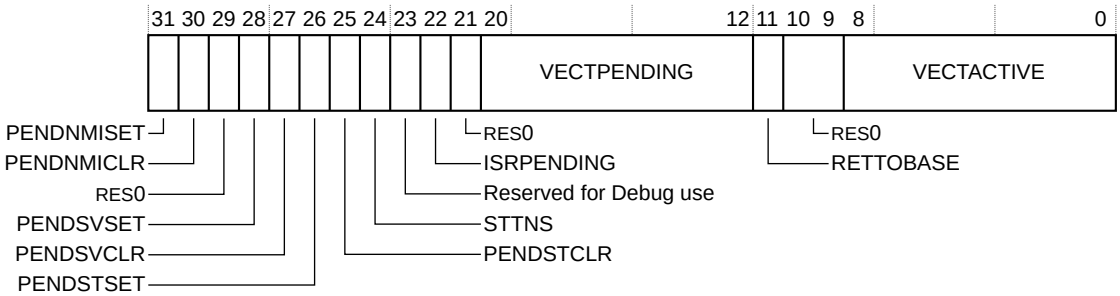


Table 5-5: ICSR bit assignments without the Security Extension

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.
[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.
[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.

Bits	Name	Type	Function
[27]	PENDSVCLR	WO	PendSV clear-pending bit. Write: **0** No effect. **1** Removes the pending state from the PendSV exception.
[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.
[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.
[24]	STTNS	RO	**RES0**.
[23]	ISRPREEMPT	RO	Interrupt preempt. Indicates whether a pending exception is handled on exit from debug state. This bit is not banked between Security states. 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.
[22]	ISRPENDING	RO	Interrupt pending flag, excluding NMI and Faults: **0** Interrupt not pending. **1** Interrupt pending.
[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.
[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.
[10:9]	-	-	Reserved, **RES0**.

Bits	Name	Type	Function
[8:0]	VECTACTIVE[16]	RO	Contains the active exception number: **0** Thread mode. **1** The exception number[16] 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 3.1.3.6.2 Interrupt Program Status Register on page 31.

Table 5-6: 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**.

[16] This is the same value as IPSR bits[8:0], see 3.1.3.6.2 Interrupt Program Status Register on page 31.

Bits	Name	Type	Function
[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.
[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**.

Bits	Name	Type	Function
[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.
[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[17]	RO	Contains the active exception number: **0** Thread mode. **1** The exception number[17] 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 3.1.3.6.2 Interrupt Program Status Register on page 31. 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.

[17] This is the same value as IPSR bits[8:0], see 3.1.3.6.2 Interrupt Program Status Register on page 31.

5.2.5 Vector Table Offset Register

The VTOR indicates the offset of the vector table base address from memory address `0x00000000`.

See 5.2.1 System control block registers summary on page 237 for the VTOR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The VTOR bit assignments are:

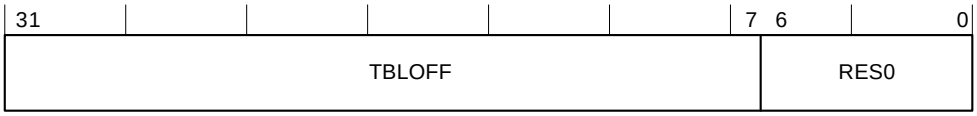


Table 5-7: VTOR bit assignments

Bits	Name	Function
[31:7]	TBLOFF	Vector table base offset field. It contains bits[31:7] of the offset of the table base from the bottom of the memory map.
[6:0]	-	Reserved, **RES0**.

When setting TBLOFF, you must align the offset to the number of exception entries in the vector table.

Table alignment requirements mean that bits[6:0] of the table offset are always zero.

5.2.6 Application Interrupt and Reset Control Register

The AIRCR provides sets or returns interrupt control and reset configuration.

See 5.2.1 System control block registers summary on page 237 for the AIRCR attributes.

To write to this register, you must write `0x5FA` to the VECTKEY field, otherwise the processor ignores the write.

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

The AIRCR bit assignments are:

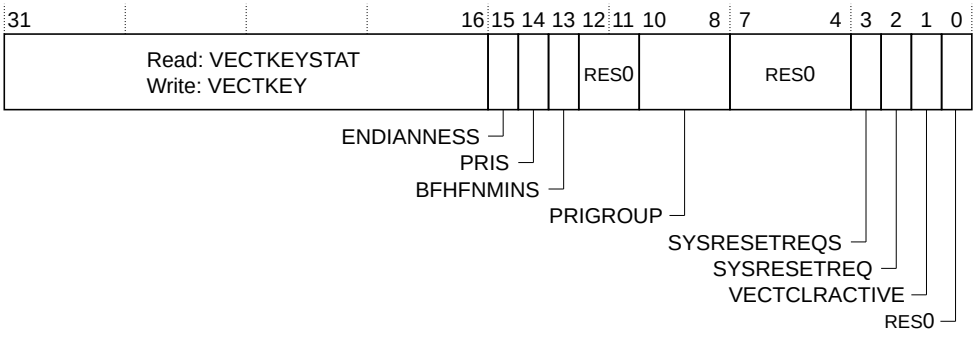


Table 5-8: AIRCR bit assignments without the Security Extension

Bits	Name	Type	Function
[31:16]	Read: VECTKEYSTAT Write: VECTKEY	RW	Register key: Reads as `0xFA05`. 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 5.2.6.1 Binary point on page 249.
[7:4]	-	-	Reserved, **RES0**.
[3]	SYSRESETREQS	RAZ/WI	-
[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 5-9: AIRCR bit assignments with the Security Extension

Bits	Name	Type	Function
[31:16]	Read: VECTKEYSTAT Write: VECTKEY	RW	Register key: Reads as `0xFA05`. 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.	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.	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**.
[10:8]	PRIGROUP	RW	Interrupt priority grouping field. This field determines the split of group priority from subpriority, see 5.2.6.1 Binary point on page 249. This bit is banked between Security states.
[7:4]	-	-	Reserved, **RES0**.
[3]	SYSRESETREQS	RW from Secure State and RAZ/WI from Non-secure state.	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.

Bits	Name	Type	Function
[2]	SYSRESETREQ	RW if SYSRESETREQS is 0. When SYSRESETREQS is set to 1, from Non-secure state this bit acts as RAZ/WI.	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**.

5.2.6.1 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 5-10: Priority grouping

	Interrupt priority level value, PRI_*n*[7:0]			Number of	
PRIGROUP	Binary point[18]	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
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.

[18] PRI_*n*[7:0] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

5.2.7 System Control Register

The SCR controls features of entry to and exit from low-power state.

See 5.2.1 System control block registers summary on page 237 for the SCR attributes.

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

The bit assignments are:

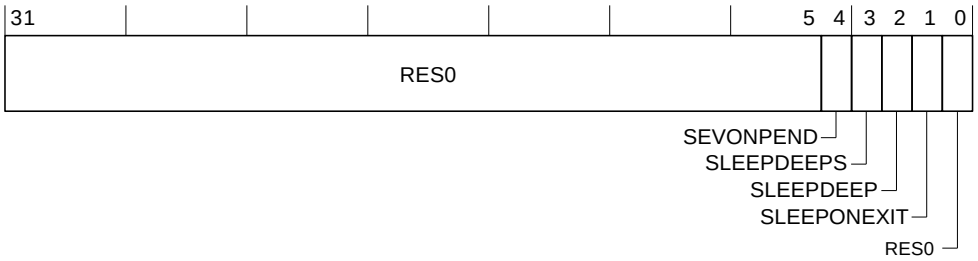


Table 5-11: SCR bit assignments without the Security Extension

Bits	Name	Function
[31:5]	-	Reserved, **RES0**.
[4]	SEVONPEND	Send Event on Pending bit: **0** Only enabled interrupts or events can wakeup the processor, disabled interrupts are excluded. **1** Enabled events and all interrupts, including disabled interrupts, can wakeup the processor. When an event or interrupt enters pending state, the event signal wakes up the processor from WFE. If the processor is not waiting for an event, the event is registered and affects the next WFE. The processor also wakes up on execution of an `SEV` instruction or an external event.
[3]	SLEEPDEEPS	RAZ/WI.
[2]	SLEEPDEEP	Controls whether the processor uses sleep or deep sleep as its low-power mode: **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 5-12: SCR bit assignments with the Security Extension

Bits	Name	Function
[31:5]	-	Reserved, **RES0**.

Bits	Name	Function
[4]	SEVONPEND	Send Event on Pending bit: **0** Only enabled interrupts or events can wakeup the processor, disabled interrupts are excluded. **1** Enabled events and all interrupts, including disabled interrupts, can wakeup the processor. When an event or interrupt enters pending state, the event signal wakes up the processor from WFE. If the processor is not waiting for an event, the event is registered and affects the next WFE. The processor also wakes up on execution of an `SEV` instruction or an external event. 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: **0** Sleep. **1** Deep sleep. This bit is not banked between Security states.
[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. This bit is banked between Security states.
[0]	-	Reserved, **RES0**.

5.2.8 Configuration and Control Register

The CCR is a read-only register and indicates some aspects of the behavior of the processor.

See 5.2.1 System control block registers summary on page 237 for the CCR attributes.

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

The bit assignments for CCR are:

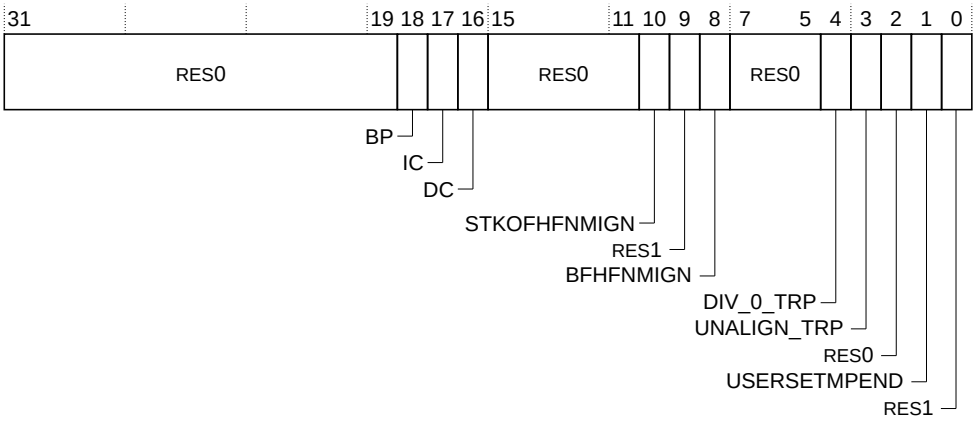


Table 5-13: CCR bit assignments without the Security Extension

Bits	Name	Function
[31:19]	-	Reserved, **RES0**
[18]	BP	RAZ/WI.
[17]	IC	RAZ/WI.
[16]	DC	RAZ/WI.
[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**.
[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 5-14: CCR bit assignments with the Security Extension

Bits	Name	Function
[31:19]	-	Reserved, **RES0**
[18]	BP	RAZ/WI.
[17]	IC	RAZ/WI.
[16]	DC	RAZ/WI.
[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**.

5.2.9 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 5.2.1 System control block registers summary on page 237 for the SHPR1-SHPR3 attributes.

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 5-15: System fault handler priority fields

Handler	Field	Register description
MemManage	PRI_4	5.2.9.1 System Handler Priority Register 1 -ARMv8M on page 254
BusFault	PRI_5	
UsageFault	PRI_6	
SecureFault	PRI_7	
SVCall	PRI_11	5.2.9.2 M33 System Handler Priority Register 2 on page 255
PendSV	PRI_14	5.2.9.3 M33 System Handler Priority Register 3 on page 255
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.

5.2.9.1 System Handler Priority Register 1

Bit assignments for the SHPR1 register.

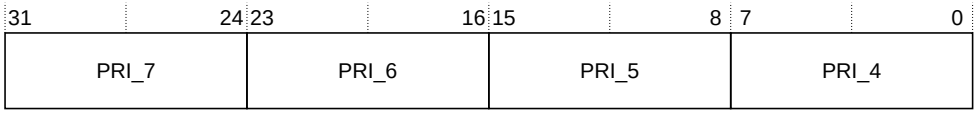


Table 5-16: SHPR1 register bit assignments

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.

5.2.9.2 System Handler Priority Register 2

Bit assignments for the SHPR2 register.

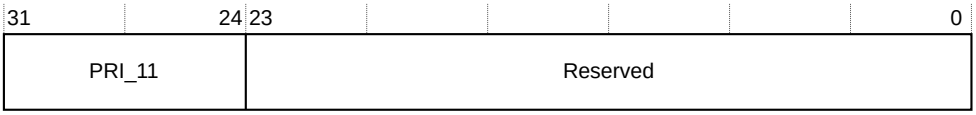


Table 5-17: SHPR2 register bit assignments

Bits	Name	Function	Security state
[31:24]	PRI_11	Priority of system handler 11, SVCall	PRI_11 is banked between Security states.
[23:0]	-	Reserved	-

5.2.9.3 System Handler Priority Register 3

Bit assignments for the SHPR3 register.

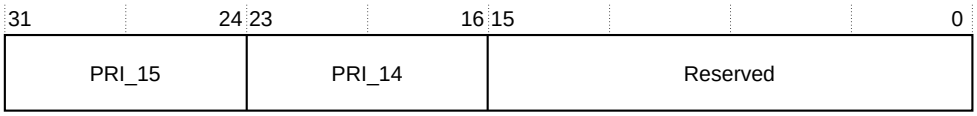


Table 5-18: SHPR3 register bit assignments

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.
[15:0]	-	Reserved	-

5.2.10 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.

See 5.2.1 System control block registers summary on page 237 for the SHCSR attributes.

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

The SHCSR bit assignments are:

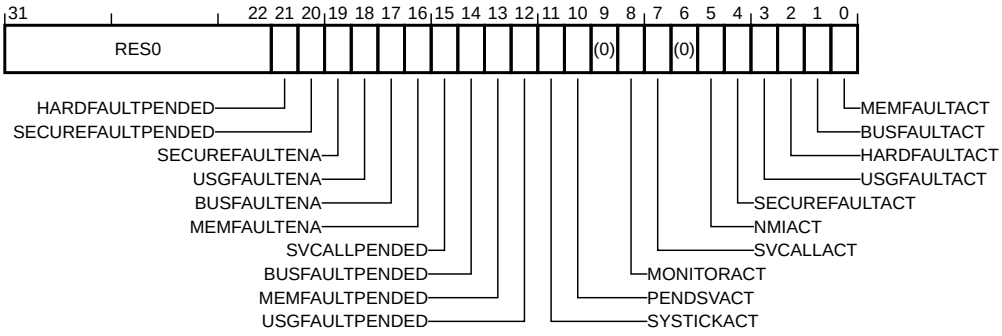


Table 5-19: SHCSR bit assignments without the Security Extension

Bits	Name	Function
[31:22]	-	Reserved, **RES0**.
[21]	HARDFAULTPENDED	HardFault exception pended state bit, set to 1 to allow exception modification
[20]	SECUREFAULTPENDED	**RES0**
[19]	SECUREFAULTENA	**RES0**
[18]	USGFAULTENA	UsageFault enable bit, set to 1 to enable.[19]
[17]	BUSFAULTENA	BusFault enable bit, set to 1 to enable.[19]
[16]	MEMFAULTENA	MemManage enable bit, set to 1 to enable. [19]
[15]	SVCALLPENDED	SVCall pending bit, reads as 1 if exception is pending. [20]
[14]	BUSFAULTPENDED	BusFault exception pending bit, reads as 1 if exception is pending. [20]
[13]	MEMFAULTPENDED	MemManage exception pending bit, reads as 1 if exception is pending.[20]
[12]	USGFAULTPENDED	UsageFault exception pending bit, reads as 1 if exception is pending.[20]
[11]	SYSTICKACT	SysTick exception active bit, reads as 1 if exception is active.[21]
[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 5-20: SHCSR bit assignments with the Security Extension

Bits	Name	Function
[31:22]	-	Reserved, **RES0**.

Bits	Name	Function
[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.
[20]	SECUREFAULTPENDED	SecureFault exception pended state bit, set to 1 to allow exception modification. This bit is not banked between Security states.
[19]	SECUREFAULTENA	SecureFault exception enable bit, set to 1 to enable. This bit is not banked between Security states.
[18]	USGFAULTENA	UsageFault enable bit, set to 1 to enable.[19] This bit is banked between Security states.
[17]	BUSFAULTENA	BusFault enable bit, set to 1 to enable.[19] 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. [19] This bit is banked between Security states.
[15]	SVCALLPENDED	SVCall pending bit, reads as 1 if exception is pending. [20] This bit is banked between Security states.
[14]	BUSFAULTPENDED	BusFault exception pending bit, reads as 1 if exception is pending. [20] 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.[20] This bit is banked between Security states.
[12]	USGFAULTPENDED	UsageFault exception pending bit, reads as 1 if exception is pending.[20] This bit is banked between Security states.
[11]	SYSTICKACT	SysTick exception active bit, reads as 1 if exception is active.[21] 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**.

Bits	Name	Function
[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.
[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.

- 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 t restores the current active status.

[19] Enable bits, set to 1 to enable the exception, or set to 0 to disable the exception.
[20] 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.
[21] 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.

- 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.

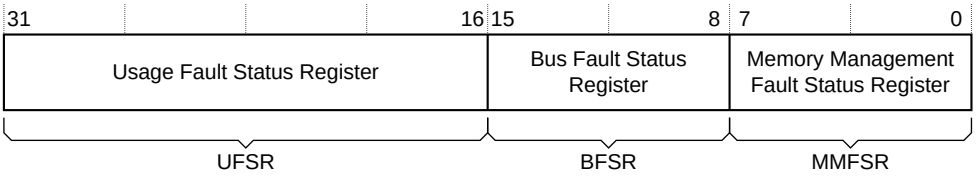
5.2.11 Configurable Fault Status Register

The CFSR indicates the cause of a MemManage fault, BusFault, or UsageFault.

See 5.2.1 System control block registers summary on page 237 for the CFSR attributes.

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

The CFSR bit assignments are:



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`.

5.2.11.1 MemManage Fault Status Register

The MMFSR is a subregister of the CFSR. The flags in the MMFSR indicate the cause of memory access faults.

In an implementation with the Security Extension, this field is banked between Security states.

The bit assignments are:

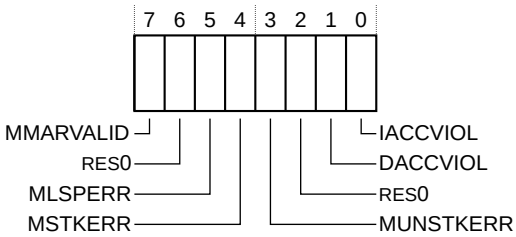


Table 5-21: MMFSR bit assignments

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.


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.

5.2.11.2 BusFault Status Register

The BFSR is a subregister of the CFSR. The flags in the BFSR indicate the cause of a bus access fault.

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 bit assignments are:

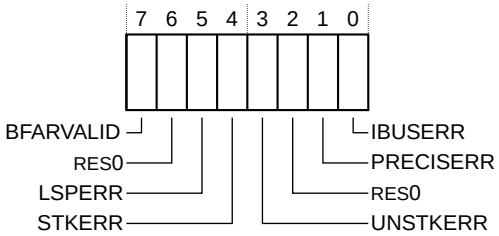


Table 5-22: BFSR bit assignments

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.

Bits	Name	Function
[2]	-	Reserved, **RES0**
[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.

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.

5.2.11.3 UsageFault Status Register

The UFSR is a subregister of the CFSR. The UFSR indicates the cause of a UsageFault.

In an implementation with the Security Extension, this field is banked between Security states.

The bit assignments are:

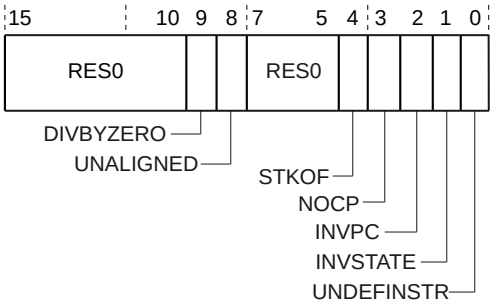


Table 5-23: UFSR bit assignments

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.
[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.

Bits	Name	Function
[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.

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.

5.2.12 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.

See 5.2.1 System control block registers summary on page 237 for the HFSR attributes.

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 HFSR bit assignments are:

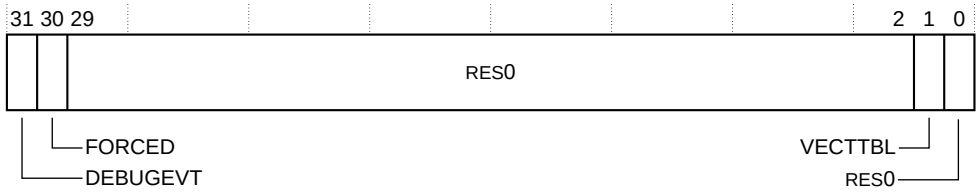


Table 5-24: HFSR bit assignments

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**.

Bits	Name	Function
[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**.
[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.

5.2.13 MemManage Fault Address Register

The MMFAR contains the address of the location that generated a MemManage fault.

See 5.2.1 System control block registers summary on page 237 for the MMFAR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The MMFAR bit assignments are:

Table 5-25: MMFAR bit assignments

Bits	Name	Function
[31:0]	ADDRESS	When the MMARVALID bit of the MMFSR is set to 1, this field holds the address of the location that generated the MemManage fault

When an unaligned access faults, the address is the actual address that faulted. Because a single read or write instruction can be split into multiple aligned accesses, the fault address can be any address in the range of the requested access size.

Flags in the MMFSR indicate the cause of the fault, and whether the value in the MMFAR is valid.

5.2.14 BusFault Address Register

The BFAR contains the address of the location that generated a BusFault.

See 5.2.1 System control block registers summary on page 237 for the BFAR attributes.

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

The BFAR bit assignments are:

Table 5-26: BFAR bit assignments

Bits	Name	Function
[31:0]	ADDRESS	When the BFARVALID bit of the BFSR is set to 1, this field holds the address of the location that generated the BusFault

When an unaligned access faults the address in the BFAR is the one requested by the instruction, even if it is not the address of the fault.

Flags in the BFSR indicate the cause of the fault, and whether the value in the BFAR is valid.

5.2.15 Coprocessor Access Control Register

The CPACR register specifies the access privileges for coprocessors.

See 5.2.1 System control block registers summary on page 237 for the CPACR attributes.

In an implementation with the Security Extension, this field is banked between Security states.

The CPACR bit assignments are:

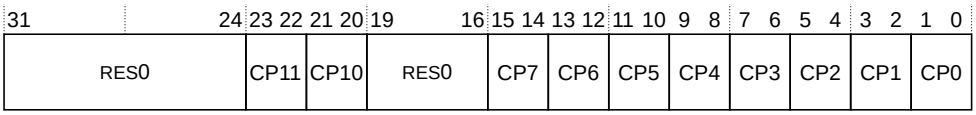


Table 5-27: CPACR bit assignments

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.

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.

5.2.16 Non-secure Access Control Register

In an implementation with the Security Extension, tThe NSACR register defines the Non-secure access permissions for both the FPU and coprocessors CP *m*, bit[*m*], for *m* = 0-7.

See the 5.2.1 System control block registers summary on page 237 for the NSACR attributes.

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

The NSACR bit assignments are:

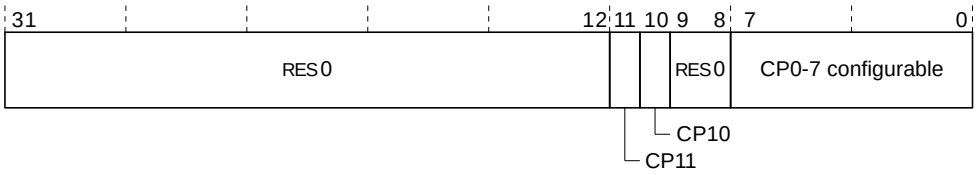


Table 5-28: NSACR bit assignments

Bits	Name	Function
[31:12]	-	Reserved, **RES0**.

Bits	Name	Function
[11]	CP11	CP11 access. Enables Non-secure access to the Floating-point Extension. Programming with a different value other than that used for CP10 is **UNPREDICTABLE**. If the Floating-point Extension is not implemented, this bit is RAZ/WI.
[10]	CP10	CP10 access. Enables Non-secure access to the Floating-point Extension. **0** Non-secure accesses to the Floating-point Extension generate a NOCP UsageFault. **1** Non-secure access to the Floating-point Extension permitted. If the Floating-point Extension is not implemented, this bit is RAZ/WI.
[9:8]	-	Reserved, **RES0**
CP*m*, bit[*m*], for *m* = 0-7	CP*m* for *m* = 0-7	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.

5.2.17 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.

In addition, the CMSIS provides a number of functions for system control, including:

Table 5-29: CMSIS function for system control

CMSIS system control function	Description
`void NVIC_SystemReset (void)`	Reset the system

5.3 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, each 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 5-30: System timer registers summary

Address	Name	Type	Reset value	Description
`0xE000E010`	SYST_CSR	RW	`0x00000000`	5.3.1 SysTick Control and Status Register on page 269.
`0xE000E014`	SYST_RVR	RW	UNKNOWN	5.3.2 SysTick Reload Value Register on page 270.
`0xE000E018`	SYST_CVR	RW	UNKNOWN	5.3.3 SysTick Current Value Register on page 271.
`0xE000E01C`	SYST_CALIB	RO	`0xC0000000` (SysTick calibration value)	5.3.4 SysTick Calibration Value Register on page 271.

5.3.1 SysTick Control and Status Register

The SYST_CSR controls and provides status date for the SysTick timer.

See 5.3 System timer, SysTick on page 269 for the SYST_CSR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The bit assignments for SYST_CSR are:

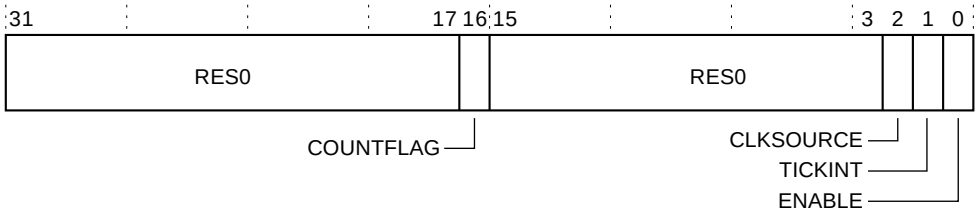


Table 5-31: SYST_CSR bit assignments

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**.
[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.

5.3.2 SysTick Reload Value Register

The SYST_RVR specifies the SysTick timer counter reload value.

See 5.3 System timer, SysTick on page 269 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:

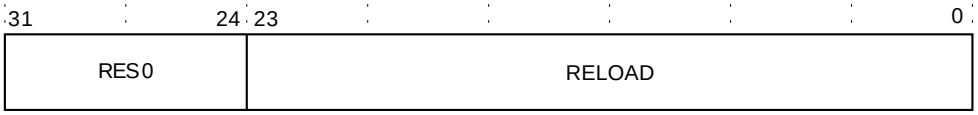


Table 5-32: SYST_RVR bit assignments

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 5.3.2.1 Calculating the RELOAD value on page 271.

5.3.2.1 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.

5.3.3 SysTick Current Value Register

The SYST_CVR contains the current value of the SysTick counter.

See 5.3 System timer, SysTick on page 269 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:

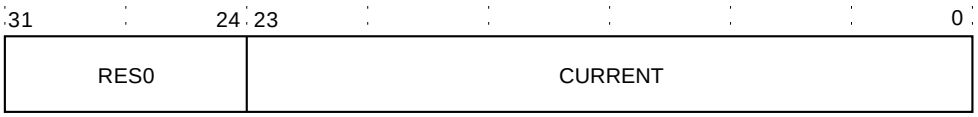


Table 5-33: SYST_CVR bit assignments

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.

5.3.4 SysTick Calibration Value Register

The SYST_CALIB register indicates the SysTick calibration value and parameters for the selected Security state.

See 5.3 System timer, SysTick on page 269 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 are:

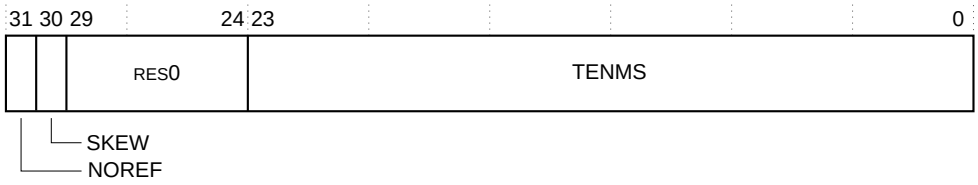


Table 5-34: SYST_CALIB bit assignments

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, or not given. 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.

If calibration information is not known, calculate the calibration value required from the frequency of the core clock or external clock.

5.3.5 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 Control and Status register.

5.4 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 level of 0-255. A higher level corresponds to a lower priority, so level 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 *Wake-up Interrupt Controller* (WIC).
- 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 5-35: NVIC registers summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E13C	NVIC_ISER0-NVIC_ISER15	RW	Privileged	0x00000000	5.4.2 Interrupt Set Enable Registers - Cortex-M33 on page 274
0XE000E180-0xE000E1BC	NVIC_ICER0-NVIC_ICER15	RW	Privileged	0x00000000	5.4.3 Interrupt Clear Enable Registers - Cortex-M33 on page 275
0XE000E200-0xE000E23C	NVIC_ISPR0-NVIC_ISPR15	RW	Privileged	0x00000000	5.4.4 Interrupt Set Pending Registers - Cortex-M33 on page 276
0XE000E280-0xE000E2BC	NVIC_ICPR0-NVIC_ICPR15	RW	Privileged	0x00000000	5.4.5 Interrupt Clear Pending Registers - Cortex-M33 on page 277
0xE000E300-0xE000E33C	NVIC_IABR0-NVIC_IABR15	RW	Privileged	0x00000000	5.4.6 Interrupt Active Bit Registers - CortexM33 on page 278
0xE000E380-0xE000E3BC	NVIC_ITNS0-NVIC_ITNS15	RW[22]	Privileged	0x00000000	5.4.7 Interrupt Target Non-secure Registers - CortexM33 on page 278.
0xE000E400-0xE000E5DC	NVIC_IPR0-NVIC_IPR119	RW	Privileged	0x00000000	5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279
0xE000EF00	STIR	WO	Configurable[23]	0x00000000	5.4.9 Software Trigger Interrupt Register on page 281

[22] ITNS is RAZ/WI from the Non-Secure state.
[23] See the register description for more information.

5.4.1 Accessing the NVIC registers using CMSIS

CMSIS functions enable software portability between different Cortex®-M profile processors.

To access the NVIC registers when using CMSIS, use the following functions:

Table 5-36: 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

5.4.2 Interrupt Set Enable Registers

The NVIC_ISER0-NVIC_ISER15 registers enable interrupts, and show which interrupts are enabled.

See the register summary in 5.4 Nested Vectored Interrupt Controller on page 272 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:

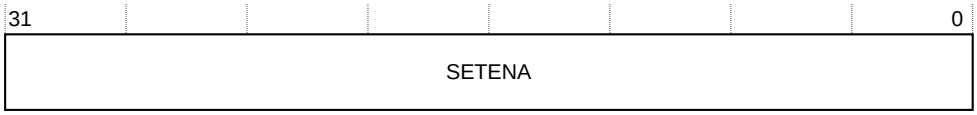


Table 5-37: NVIC_ISERn bit assignments

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.

5.4.3 Interrupt Clear Enable Registers

The NVIC_ICER0-NVIC_ICER15 registers disable interrupts, and show which interrupts are enabled.

See the register summary in 5.4 Nested Vectored Interrupt Controller on page 272 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:

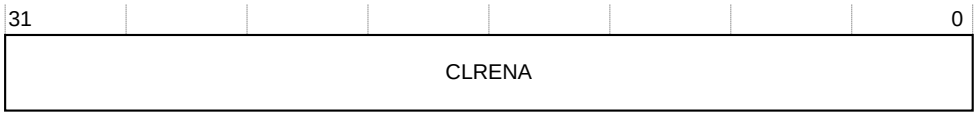


Table 5-38: NVIC_ICERn bit assignments

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.

5.4.4 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 5.4 Nested Vectored Interrupt Controller on page 272 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 5-39: 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.

5.4.5 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 5.4 Nested Vectored Interrupt Controller on page 272 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.

The bit assignments are:

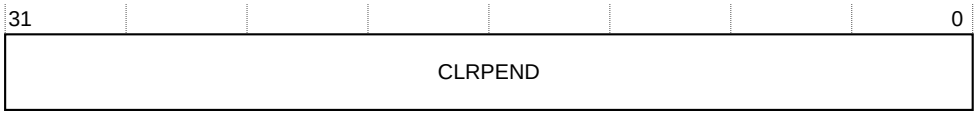


Table 5-40: NVIC_ICPRn bit assignments

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.

5.4.6 Interrupt Active Bit Registers

The NVIC_IABR0-NVIC_IABR15 registers indicate the active state of each interrupt.

See the register summary in 5.4 Nested Vectored Interrupt Controller on page 272 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 5-41: 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.

5.4.7 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 5.4 Nested Vectored Interrupt Controller on page 272 for the register attributes.

In an implementation with the Security Extension, this register is accessible from Secure state only.

The bit assignments are:

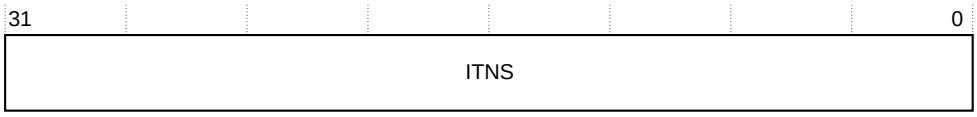


Table 5-42: NVIC_ITNSn bit assignments

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.

5.4.8 Interrupt Priority Registers

The NVIC_IPR0-NVIC_IPR119 registers provide an 8-bit priority field for each interrupt. These registers are word, halfword, and byte accessible.

See the register summary in 5.4 Nested Vectored Interrupt Controller on page 272 for their attributes.

Each register holds four priority fields as shown:

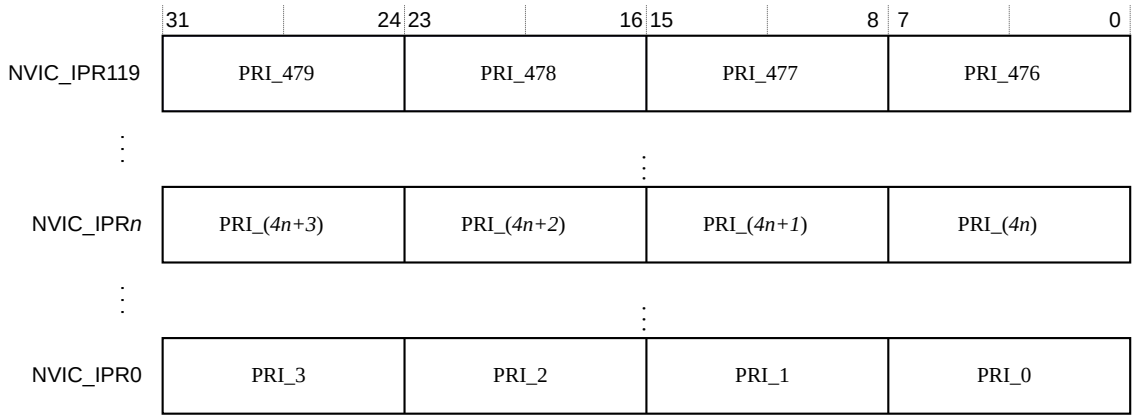


Table 5-43: NVIC_IPRn bit assignments

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. If the processor implements fewer than 8 bits of priority, then the least significant bits of this field are **RES0**.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

See 5.4.1 Accessing the NVIC registers using CMSIS on page 274 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.

5.4.9 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.

Only privileged software can enable unprivileged access to the STIR.

See 5.4 Nested Vectored Interrupt Controller on page 272 for the register attributes.

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

The bit assignments are:

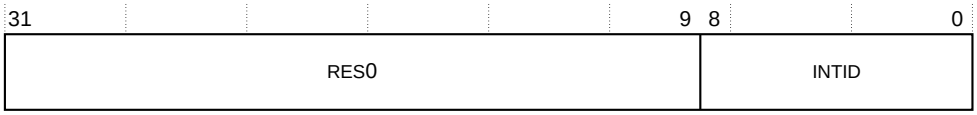


Table 5-44: STIR bit assignments

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.

5.4.10 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.

5.4.10.1 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 Interrupt Set Enable Register 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 Interrupt Clear Pending Register 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.

5.4.11 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.

5.4.11.1 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 5.4.1 Accessing the NVIC registers using CMSIS on page 274.

The input parameter `IRQn` is the IRQ number, see 3.3.2 se_Exception types on page 51 for more information. For more information about these functions, see the CMSIS documentation.

5.5 Security Attribution and Memory Protection

If the Security Extension is implemented, the processor can use security attribution and memory protection to manage sensitive data.

The processor can have an *Security Attribution Unit* (SAU) and a *Memory Protection Unit* (MPU) that provide fine grain memory control, enabling applications to use multiple privilege levels, separating and protecting code, data, and stack on a task-by-task basis. Such requirements are becoming critical in many embedded applications such as automotive systems.

Some implementations might only have one MPU.

5.5.1 Security Attribution Unit

The SAU determines the security of an address.

For instructions, the SAU returns the security attribute (Secure or Non-secure) and identifies whether the instruction address is in a Non-secure callable region.

For data, the SAU returns the security attribute (Secure or Non-secure).

When a memory access is performed, the security of the address is verified by the SAU. Any address that matches multiple SAU regions will be marked with the most secure attribute of the matching regions.

The following table shows a summary of the SAU registers.

Table 5-45: SAU registers summary

Address	Name	Type	Reset value	Description
0xE000EDD0	SAU_CTRL	RW	0x00000000	See 5.5.2 Security Attribution Unit Control Register on page 284. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI.
0xE000EDD4	SAU_TYPE	RO	0x00000000	See 5.5.3 Security Attribution Unit Type Register on page 285. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI. SAU_TYPE [7:0] reflects the number of SAU regions.
0xE000EDD8	SAU_RNR	RW	**UNKNOWN**	See 5.5.4 Security Attribution Unit Region Number Register on page 286. In Non-secure state, this register is RAZ/WI.
0xE000EDDC	SAU_RBAR	RW	**UNKNOWN**	See 5.5.5 Security Attribution Unit Region Base Address Register on page 286. 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 5.5.6 Security Attribution Unit Region Limit Address Register on page 287. This is the reset value in Secure state. In Non-secure state, this register is RAZ/WI.
0xE000EDE4	SFSR	RW	0x00000000	See 5.5.7 Secure Fault Status Register on page 288. In Non-secure state, this register is RAZ/WI.
0xE000EDE8	SFAR	RW	**UNKNOWN**	See 5.5.8 Secure Fault Address Register on page 289. 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.

5.5.2 Security Attribution Unit Control Register

The SAU_CTRL allows enabling of the Security Attribution Unit.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SAU_CTRL bit assignments are:

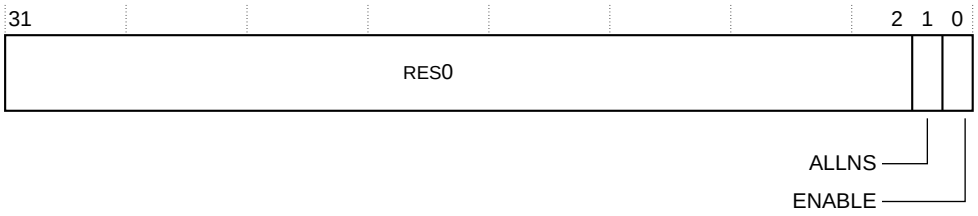


Table 5-46: SAU_CTRL bit assignments

Bits	Name	Function
[31:2]	-	Reserved, **RES0**.
[1]	ALLNS	All Non-secure. When SAU_CTRL.ENABLE is 0 this bit controls if the memory is marked as Non-secure or Secure. The possible values of this bit are: **0** Memory is marked as Secure and is not Non-secure callable. **1** Memory is marked as Non-secure. This bit has no effect when SAU_ENABLE is 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	Enable. Enables the SAU. The possible values of this bit are: **0** The SAU is disabled. **1** The SAU is enabled. This bit is RAZ/WI when the Security Extension is implemented without an SAU region.

5.5.3 Security Attribution Unit Type Register

The SAU_TYPE indicates the number of regions implemented by the Security Attribution Unit.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SAU_TYPE bit assignments are:

Table 5-47: SAU_TYPE bit assignments

Bits	Name	Function
[31:8]	-	Reserved, **RES0**.

Bits	Name	Function
[7:0]	SREGION	SAU regions. The number of implemented SAU regions.

5.5.4 Security Attribution Unit Region Number Register

The SAU_RNR selects the region currently accessed by SAU_RBAR and SAU_RLAR.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SAU_RNR bit assignments are:

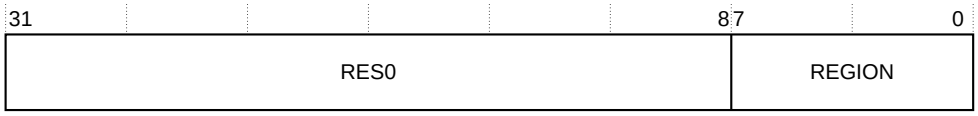


Table 5-48: SAU_RNR bit assignments

Bits	Name	Function
[31:8]	-	Reserved, **RES0**.
[7:0]	REGION	Region number. Indicates the SAU region accessed by SAU_RBAR and SAU_RLAR. If no SAU 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.

5.5.5 Security Attribution Unit Region Base Address Register

The SAU_RBAR provides indirect read and write access to the base address of the currently selected SAU region.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SAU_RBAR bit assignments are:

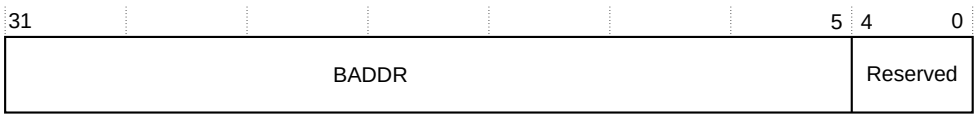


Table 5-49: SAU_RBAR bit assignments

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**.

5.5.6 Security Attribution Unit Region Limit Address Register

The SAU_RLAR provides indirect read and write access to the limit address of the currently selected SAU region.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SAU_RLAR bit assignments are:

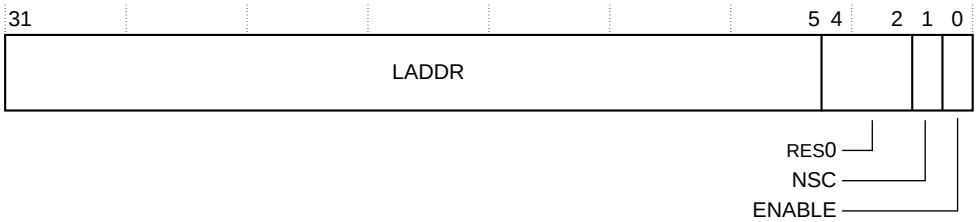


Table 5-50: SAU_RLAR bit assignments

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** SAU region is disabled. **1** SAU region is enabled. This bit reset to an **IMPLEMENTATION DEFINED** value on a Warm reset.

5.5.7 Secure Fault Status Register

The SFSR provides information about any security related faults.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

See 5.2.1 System control block registers summary on page 237 for the SFSR attributes.

The SFSR bit assignments are:

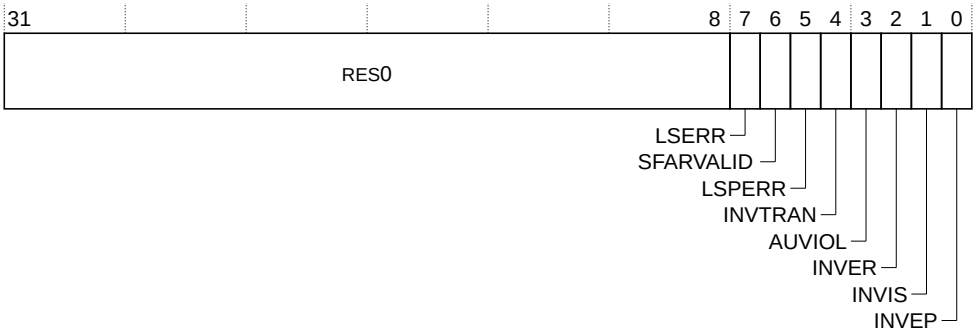


Table 5-51: SFSR bit assignments

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 SAU or IDAU violation occurred during the lazy preservation of floating-point state. The possible values of this bit are: **0** Error has not occurred. **1** Error has occurred.
[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.

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.

5.5.8 Secure Fault Address Register

The SFSR shows the address of the memory location that caused a security violation.

In an implementation with the Security Extension, this register is:

- RAZ/WI when accessed as Non-secure.
- Not banked between Security states.

The SFAR bit assignments are:

Table 5-52: 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 an SAU violation.

5.5.9 Memory Protection Unit

The MPU is divided into eight 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 3.2.2 Memory regions, types, and attributes on page 41.

The following table shows the possible MPU region attributes. These include Shareability and cache behavior attributes that are not relevant to most microcontroller implementations.

See 5.5.20.1 MPU configuration for a microcontroller on page 299 for guidelines for programming such an implementation.

Table 5-53: 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.

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.

Use the MPU registers to define the MPU regions and their attributes.

The following table shows a summary of the MPU registers.

Table 5-54: MPU registers summary

Address	Name	Type	Reset Value	Description
0xE000ED90	MPU_TYPE	RO	The reset value is fixed and depends on the value of bits[15:8] and implementation options.	See 5.5.10 MPU Type Register on page 291.
0xE000ED94	MPU_CTRL	RW	0x00000000	See 5.5.11 MPU Control Register on page 292.
0xE000ED98	MPU_RNR	RW	**UNKNOWN**	See 5.5.12 MPU Region Number Register on page 293.
0xE000ED9C	MPU_RBAR	RW	**UNKNOWN**	See 5.5.13 MPU Region Base Address Register on page 294.
0xE000EDA0	MPU_RLAR	RW	**UNKNOWN**	See 5.5.16 MPU Region Limit Address Register on page 295.
0xE000EDA4	MPU_RBAR_A<n>	RW	**UNKNOWN**	See 5.5.14 MPU Region Base Address Register Alias, n=1-3 on page 295
0xE000EDA8	MPU_RLAR_A<n>	RW	**UNKNOWN**	See 5.5.15 MPU Region Limit Address Register Alias, n=1-3 on page 295.
0xE000EDC0	MPU_MAIR0	RW	**UNKNOWN**	See 5.5.17 MPU Memory Attribute Indirection Registers 0 and 1 on page 296.
0xE000EDC4	MPU_MAIR1	RW	**UNKNOWN**	

5.5.10 MPU Type Register

The MPU_TYPE register indicates whether the MPU is present, and if so, how many regions it supports.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_TYPE bit assignments are:

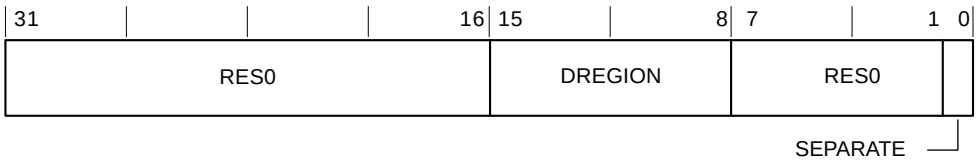


Table 5-55: MPU_TYPE bit assignments

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. **0x08** Eight 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. Arm®v8-M only supports unified MPU regions. **0** Unified.

5.5.11 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.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_CTRL bit assignments are:

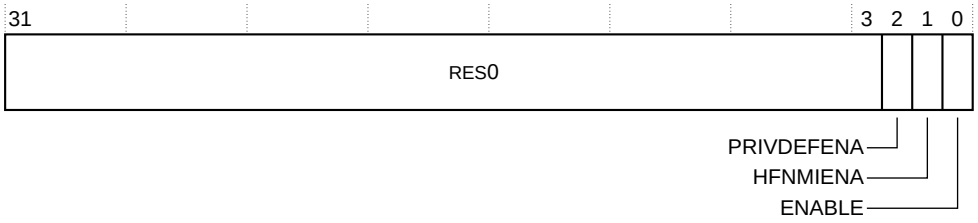


Table 5-56: MPU_CTRL bit assignments

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.

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.

5.5.12 MPU Region Number Register

The MPU_RNR selects the region currently accessed by MPU_RBAR and MPU_RLAR.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_RNR bit assignments are:

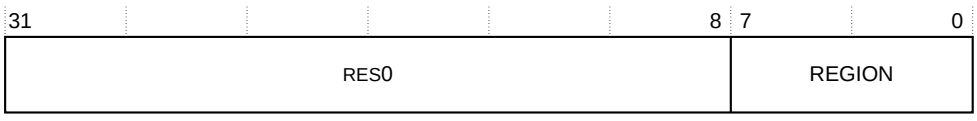


Table 5-57: MPU_RNR bit assignments

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.

5.5.13 MPU Region Base Address Register

The MPU_RBAR defines the base address of the MPU region selected by the MPU_RNR.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_RBAR bit assignments are:

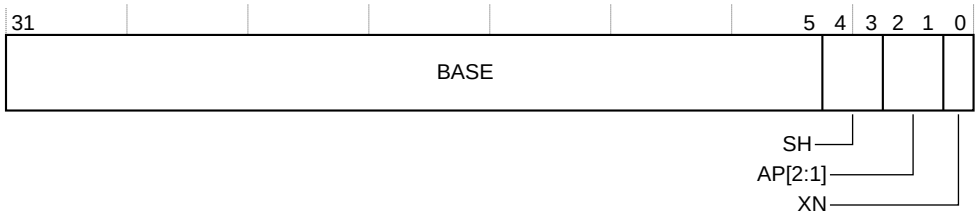


Table 5-58: MPU_RBAR bit assignments

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.
[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.

Bits	Name	Function
[0]	XN	Execute Never. Defines whether code can be executed from this region. **0** Execution only permitted if read permitted. **1** Execution not permitted.

5.5.14 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.

5.5.15 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

5.5.16 MPU Region Limit Address Register

The MPU_RLAR defines the limit address of the MPU region selected by the MPU_RNR.

In an implementation with the Security Extension, this register is banked between Security states.

The MPU_RLAR bit assignments are:

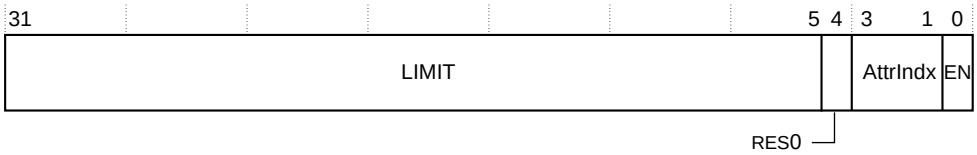


Table 5-59: MPU_RLAR bit assignments

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]	-	Reserved, **RES0**.
[3:1]	AttrIndx	Attribute index. Associates a set of attributes in the MPU_MAIR0 and MPU_MAIR1 fields.

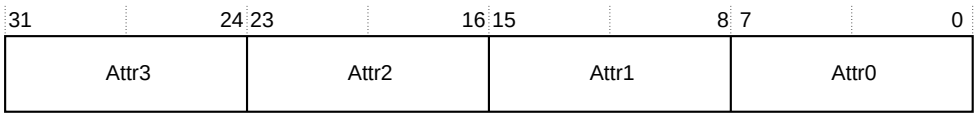
Bits	Name	Function
[0]	EN	Enable. Region enable. The possible values of this bit are: **0** Region disabled. **1** Region enabled.

5.5.17 MPU Memory Attribute Indirection Registers 0 and 1

The MPU_MAIR0 and MPU_MAIR1 provide the memory attribute encodings corresponding to the AttrIndex values.

In an implementation with the Security Extension, these registers are is banked between Security states.

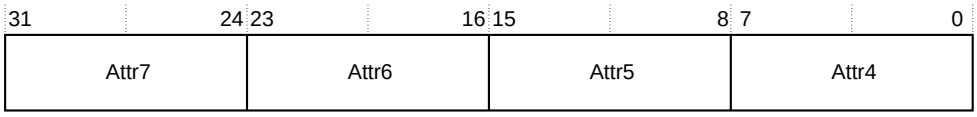
The MPU_MAIR0 bit assignments are:



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:



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`:

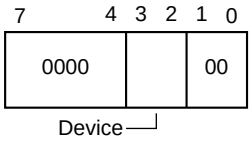


Table 5-60: MAIR_ATTR values for bits[3:2] when MAIR_ATTR[7:4] is 0000

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`:

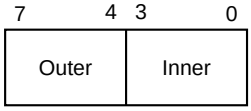


Table 5-61: MAIR_ATTR bit assignments when MAIR_ATTR[7:4] is not 0000

Bits	Name	Function
[7:4]	Outer	Outer attributes. Specifies the Outer memory attributes. The possible values of this field are: **0b0000** Device memory. In this case, refer to 5.5.17 MPU Memory Attribute Indirection Registers 0 and 1 on page 296. **00RW** Normal memory, Outer write-through transient (RW is not `00`). **0b0100** Normal memory, Outer non-cacheable. **01RW** Normal memory, Outer write-back transient (RW is not `00`). **10RW** Normal memory, Outer write-through non-transient. **11RW** Normal memory, Outer write-back non-transient. R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

Bits	Name	Function
[3:0]	Inner	Inner attributes. Specifies the Inner memory attributes. The possible values of this field are: **0b0000** **UNPREDICTABLE**. **00RW** Normal memory, Inner write-through transient (RW is not 00). **0b0100** Normal memory, Inner non-cacheable. **01RW** Normal memory, Inner write-back transient (RW is not 00). **10RW** Normal memory, Inner write-through non-transient. **11RW** Normal memory, Inner write-back non-transient. R and W specify the outer read and write allocation policy: 0 = do not allocate, 1 = allocate.

5.5.18 MPU mismatch

When access violates the MPU permissions, the processor generates a MemManage fault.

5.5.19 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.

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.

5.5.20 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.

5.5.20.1 MPU configuration for a microcontroller

Usually, a microcontroller system has only a single processor and no caches.

In such a system, program the MPU as follows:

Table 5-62: Memory region attributes for a microcontroller

Memory region	MAIR_ATTR.Outer MAIR_ATTRInner	Shareability	Memory type and attributes
Flash memory	`0b1010`	0	Normal memory, Non-shareable, Write-Through.
Internal SRAM	`0b1010`	1	Normal memory, Shareable, Write-Through.
External SRAM	`0b1111`	1	Normal memory, Shareable, Write-Back, write-allocate.
Peripherals	`0b0000`	-	Always Shareable.

In most microcontroller implementations, the cache policy attributes do not affect the system behavior. 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.

5.6 Floating-Point Unit

The Cortex®-M33 *Floating-Point Unit* (FPU) implements the FPv5 floating-point extensions.The FPU fully supports single-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. It also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions.

The FPU provides floating-point computation functionality that is compliant with the *ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic*, referred to as the IEEE 754 standard.

The FPU contains 32 single-precision extension registers, which you can also access as 16 doubleword registers for load, store, and move operations.

5.6.1 Floating-point Context Control Register

The FPCCR register sets or returns FPU control data.

See 5.6 Floating-Point Unit on page 300 for the FPCCR attributes.

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

The FPCCR bit assignments are:

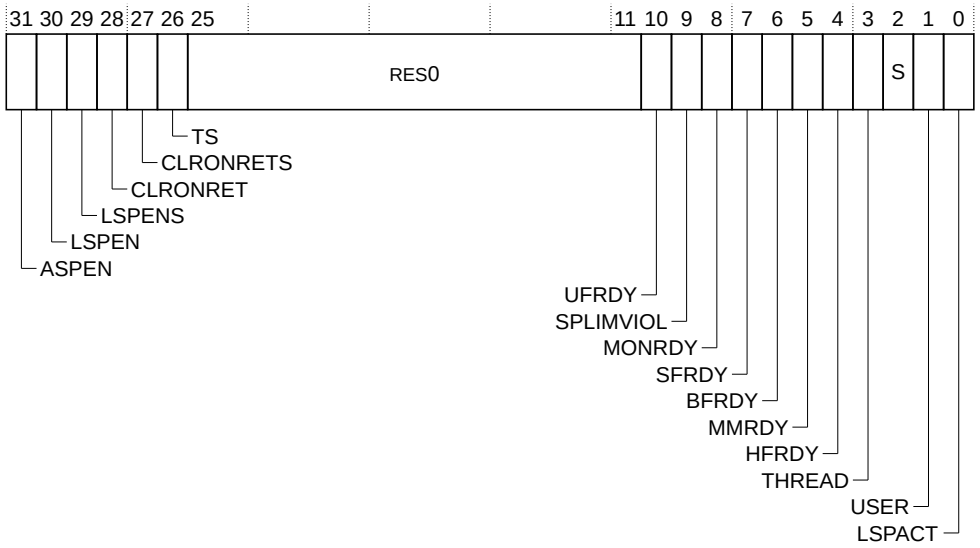


Table 5-63: FPCCR bit assignments without the Security Extension

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**

Bits	Name	Function
[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.
[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.

Bits	Name	Function
[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.
[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 5-64: 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.

Bits	Name	Function
[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.
[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.

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.

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.

5.6.2 Floating-point Context Address Register

The FPCAR register holds the location of the unpopulated floating-point register space that is allocated on an exception stack frame.

See 5.6 Floating-Point Unit on page 300 for the FPCAR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The FPCAR bit assignments are:

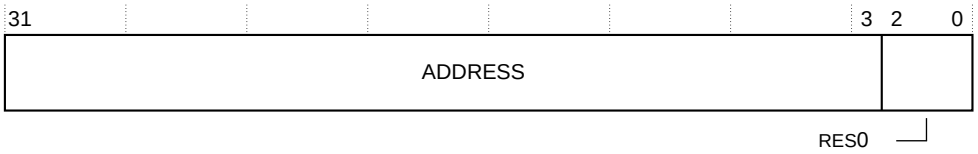


Table 5-65: FPCAR bit assignments

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**

5.6.3 Floating-point Status Control Register

The FPSCR register provides all necessary User level control of the floating-point system.

See 5.6 Floating-Point Unit on page 300 for the FPSCR attributes.

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

The FPSCR bit assignments are:

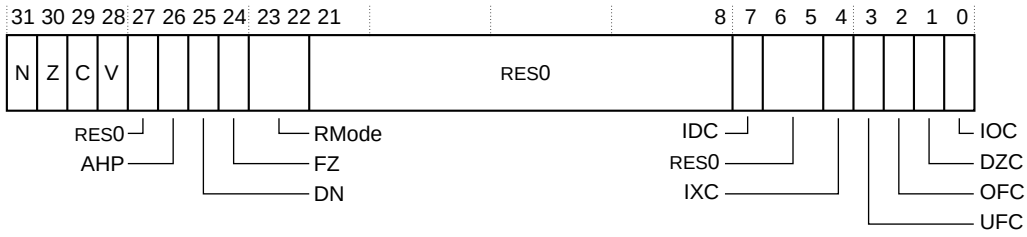


Table 5-66: FPSCR bit assignments

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.
[21:8]	-	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	

5.6.4 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.

See 5.6 Floating-Point Unit on page 300 for the FPDSCR attributes.

In an implementation with the Security Extension, this register is banked between Security states.

The FPDSCR bit assignments are:

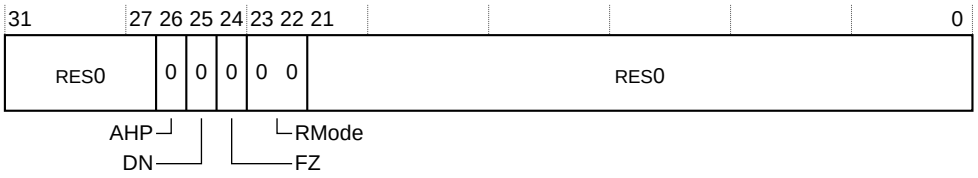


Table 5-67: FPDSCR bit assignments

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
[21:0]	-	Reserved, **RES0**

5.6.5 Code sequence for enabling the FPU

The FPU is disabled from reset. You must enable it before you can use any floating-point instructions. The code sequence shows how to a enable the FPU in privileged mode. The core must be in privileged mode to read from and write to the CPACR.

If the Security Extension is implemented, when the system boots up, the secure software should setup NSACR to determine if the FPU (coprocessor 10 and 11) is accessible from Non-secure side. The Secure software should also configure FPCCR to determine if the FPU is used by Secure software. After that, the FPU can be enabled.

Enabling the FPU

```
CPACR   EQU  0xE000ED88
LDR     R0,  =CPACR          ; Read CPACR
LDR     r1, [R0]             ; Set bits 20-23 to enable CP10 and CP11
 coprocessors
ORR     R1, R1, #(0xF << 20)
STR     R1, [R0]             ; Write back the modified value to the CPACR
DSB
```

```
ISB                             ; Reset pipeline now the FPU is enabled.
```

Appendix A Processor options

This appendix describes what the configuration options are and the affect these have on this book. The configuration options for a Cortex®-M33 processor implementation are determined by the device manufacturer.

A.1 Processor implementation options

The following table shows the processor implementation options.

Table A-1: Effects of the processor implementation options

Option	Description and affected documentation
RTL version	This affects the availability of some features. This affects: • Variant and Revision field values in 5.2.3 CPUID Base Register - ARMv8M on page 239. • The CPUID Register reset value in 5.2 System Control Block on page 237.
Inclusion of DSP Extension	The SoC designer decides whether to implement the processor with or without the DSP Extension. This affects references to the DSP Extension in: • 2.1 About the Cortex-M33 processor and core peripherals on page 18 • 4.4 General data processing instructions on page 101 • 4.7 Multiply and divide instructions on page 141 • 4.8 Saturating instructions on page 156 • 4.9 Packing and unpacking instructions on page 165
Inclusion of coprocessor	The SoC designer decides whether to implement the processor with or without a coprocessor. This affects references to the coprocessor in: • 4.5 Coprocessor instructions on page 131 • 4.6 CDE instructions on page 134 This also affects the: • 5.2.15 Coprocessor Access Control Register - ARMv8M on page 266 • 5.2.16 Non-secure Access Control Register on page 267 • 3.5.1 Fault types reference table on page 69 • 5.2.11.3 UsageFault Status Register on page 262
Inclusion of debug	The SoC designer decides whether to implement the processor with or without debug. The number of breakpoints and watchpoints is configurable to 0, 4 or 8. This affects references to the coprocessor in: • 2.1 About the Cortex-M33 processor and core peripherals on page 18. • 2.1.3 Integrated configurable debug on page 22. • 2.1.4 Processor features and benefits summary on page 22. • 3.1.3 Core registers on page 25. • 3.5.4 Lockup on page 73.

Option	Description and affected documentation
Inclusion of MPU	The SoC designer decides whether to implement the processor with or without a *Memory Protection Unit* (MPU). The number of MPU regions is configurable to 0, 4, 8, 12, or 16. This affects references to the MPU or MPU registers in: • 2.1 About the Cortex-M33 processor and core peripherals on page 18 • 3.2.2 Memory regions, types, and attributes on page 41. • 3.2.5 Behavior of memory accesses on page 44 • 3.3.2 se_Exception types on page 51 in the description of MemManage. • 3.5 Fault handling on page 69. • 5.1 About the Cortex-M33 peripherals on page 236. Include either: ◦ The row for `0xE000ED90`, MPU Type Register, reads as zero. ◦ The row for `0xE000ED90-0xE000EDB8`, Memory Protection Unit. If you have cache in your memory system, this affects bit field information in Table 5-60: MAIR_ATTR values for bits[3:2] when MAIR_ATTR[7:4] is 0000 on page 297
Inclusion of FPU	The SoC designer decides whether to implement the processor with or without a single-precision *Floating-Point Unit* (FPU). This affects: • 4.12 Floating-point instructions on page 178. • The inclusion of VLDM/VSTM/VPUSH/VPOP in the list of interruptible instructions 3.1.3.6.4 Interruptible-continuable instructions on page 33. • The FPCA bit in 3.1.3.8 CONTROL register on page 38. • The MLSPERR bit in the *MemManage Fault Status Register* (MMFSR). • The LSPERR and LSERR bits in the *SecureFault Status Register* (SFSR) if the Security Extension is included.
Number of interrupts	The SoC designer decides how many interrupts your processor implementation supports, in the range 1-480. This affects: • The maximum value of ISR_NUMBER in 3.1.3.6.2 Interrupt Program Status Register on page 31. • Exception number values (16 and above) in 3.3.2 se_Exception types on page 51, particularly if you implement only one. • The maximum interrupt number, and associated information where appropriate, in: ◦ 3.3.3 se_Exception handlers on page 56. ◦ 3.3.4 se_Vector table on page 57 ◦ 5.4 Nested Vectored Interrupt Controller on page 272 • The number of implemented *Nested Vectored Interrupt Controller* (NVIC) registers in: ◦ NVIC register summary ◦ The appropriate register descriptions in sections 5.4.2 Interrupt Set Enable Registers - Cortex-M33 on page 274 to 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279 • 5.2.5 Vector Table Offset Register on page 245.

Option	Description and affected documentation
Number of priority bits	The SoC designer decides how many priority bits are in priority value fields, in the range 3-8. Register priority value fields are 8 bits wide, and unimplemented low-order bits read as zero and ignore writes. This affects: • The note in 3.3.5 se_Exception priorities on page 59 • The notes in 3.1.3.8 CONTROL register on page 38 • The maximum priority level value in the introduction to 5.4 Nested Vectored Interrupt Controller on page 272 • In 5.4.8 Interrupt Priority Registers - Cortex-M33 on page 279 ◦ The maximum priority level value, in the introductory sentence. ◦ The priority field description, in 5.4.7 Interrupt Target Non-secure Registers - CortexM33 on page 278 • In 5.2.9 M33 System Handler Priority Registers on page 253: ◦ The field width, in the introductory sentence. ◦ The priority fields description in 5.2.9.3 M33 System Handler Priority Register 3 on page 255 ◦ The description of the effect of the binary point, in 5.2.6.1 Binary point on page 249.
Inclusion of the WIC	The SoC designer decides whether to implement the processor with or without a *Wakeup Interrupt Controller* (WIC). This affects references to the WIC in: • 2.1 About the Cortex-M33 processor and core peripherals on page 18. • 3.6 Power management on page 73. • 3.6.3 The Wakeup Interrupt Controller on page 75.
Sleep mode power-saving	The SoC designer decides the power-saving options available in the sleep modes. This affects 3.6 Power management on page 73. Sleep mode power saving might also affect SysTick behavior, and you might have to revise the description inwhich affects 5.3.5 SysTick usage hints and tips on page 272.
Endianness	The implementer decides whether the memory system is little-endian or big-endian. This affects: • Descriptions of endianness in: ◦ 3.1.5 Data types and data memory accesses on page 39. ◦ The introductory paragraph in 3.2.7 Memory endianness on page 46. Include either 3.2.7.1 Byte-invariant big-endian format on page 46 or 3.2.7.2 Little-endian format on page 47. but not both.
Memory features	Some features of the memory system are implementation-specific. This affects details of vendor-specific memory in 3.2 Memory model on page 40, including: • Implementation in 2.1 About the Cortex-M33 processor and core peripherals on page 18 • 3.2.5 Behavior of memory accesses on page 44
VTOR.TBLOFF[31:7] vector base address	The SoC Designer decides the initial value in the *Vector Table Offset Register* (VTOR), which controls the vector base address. This affects the address from where the processor loads: • The MSP value in 3.1.3.2 Stack Pointer on page 27. • The PC value in 3.1.3.5 Program Counter on page 29.

Option	Description and affected documentation
Inclusion of Arm®v8-M Security Extension	The SoC designer decides whether to implement the processor with or without the Security Extension. This affects: • Figure 1-1 Cortex®-M33 processor implementation in Processor implementation on page 18 • Security Extension: ◦ 2.1.2 Security Extension on page 21. ◦ 3.1.2 Security states on page 24. ◦ 3.2.4 Secure memory system and memory partitioning on page 43. • Exception types, Secure HardFault and SecureFault in: ◦ IPSR bit assignments in 3.1.3.6.2 Interrupt Program Status Register on page 31. ◦ Properties of the different exception types Reset, NMI, HardFault, Secure HardFault, and SecureFault in 3.3 se_Exception model on page 50. • Stack pointer. 3.1.3.2 Stack Pointer on page 27. • Vector table offset. 3.3.4 se_Vector table on page 57. • System timer. 5.3 System timer, SysTick on page 269. • PRIMASK, FAULTMASK, and BASEPRI registers, in 3.1.3.7 Exception mask registers on page 34 • MPU: ◦ There can be two MPUs, one Secure and one Non-secure. Each MPU can define memory attributes independently.' 2.1.5 Cortex-M33 Processor core peripherals on page 22. ◦ Include or omit 5.5 Security Attribution and Memory Protection on page 283 • SAU: ◦ 5.5 Security Attribution and Memory Protection on page 283.

Appendix B Revisions

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

B.1 Revisions

This section describes the technical changes between released issues of this document.

Table B-1: Issue 0002-00

Change	Location
First Non-Confidential release for r0p2	-

Table B-2: Differences between issue 0002-00 and issue 0003-00

Change	Location
First Non-Confidential release for r0p3	-
Updated CPUID reset value	5.2.1 System control block registers summary on page 237 5.2.3 CPUID Base Register - ARMv8M on page 239
Replaced *Updating MPU regions* with *Updating protected memory regions*, which includes updating SAU and MPU descriptions	5.5.19 Updating protected memory regions on page 298

Table B-3: Differences between issue 0003-00 and issue 0004-00

Change	Location
First Non-Confidential release for r0p4	-
Clarified function of the interrupt clear-enable bits.	5.4.3 Interrupt Clear Enable Registers - Cortex-M33 on page 275
Updated CPUID reset value.	5.2.1 System control block registers summary on page 237 5.2.3 CPUID Base Register - ARMv8M on page 239
Changed 'INITSVTOR pin' and 'INITNSVTOR pin' to 'INITSVTOR bus' and 'INITNSVTOR bus' where applicable.	3.3.4 se_Vector table on page 57

Table B-4: Differences between issue 0004-00 and issue 0100-00

Change	Location
First Non-Confidential release for r1p0	-
Updated CPUID reset value	5.2.1 System control block registers summary on page 237 5.2.3 CPUID Base Register - ARMv8M on page 239
Added CDE instructions	4.6 CDE instructions on page 134
Updated description of bits [31:30] in ATCLR register description	5.2.2 Auxiliary Control Register - Cortex-M33 on page 238

Change	Location
VTOR description about register banking corrected	5.2.5 Vector Table Offset Register on page 245
Type and function descriptions for SYSRESETREQ bit in AIRCR is corrected.	5.2.6 Application Interrupt and Reset Control Register - ARMv8 on page 246
Clarified MREGION field conditions	4.13.14 TT, TTT, TTA, and TTAT on page 215

Table B-5: Differences between issue 0100-00 and issue 0100-05

Change	Location
Second Non-Confidential release for r1p0	-
Editorial changes	Throughout document
Updated VTOR bit assignments Function desciption	5.2.5 Vector Table Offset Register on page 245
Corrected SADD16 and SADD8 operation instructions and condition flags	4.4.10 SADD16 and SADD8 on page 113
Corrected SASX and SSAX operation instructions and condition flags	4.4.11 SASX and SSAX on page 114
Updated SAU_RLAR bit assignments	5.5.6 Security Attribution Unit Region Limit Address Register on page 287
Changed topic title 'Additional reading' to 'Useful resources'	1.4 Useful resources on page 16