ARM® Cortex®-M7 Devices

Generic User Guide

ARM®

ARM Cortex-M7 Devices
Generic User Guide

Copyright © 2015 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Proprietary Notice

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.

Product Status

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

Web Address

http://www.arm.com

Contents
ARM Cortex-M7 Devices Generic User Guide

Preface

This preface introduces the *Cortex®-M7 Devices Generic User Guide*. It contains the following sections:

- *About this book* on page vii.
- *Feedback* on page x.

About this book

This book is a generic user guide for devices that implement the ARM Cortex-M7 processor. Implementers of Cortex-M7 designs make a number of implementation choices, that can affect the functionality of the device. This means that, in this book:

- Some information is described as implementation defined. This can be a function, mode, range, value or setting that an ARM partner can choose to configure at implementation.
- Some features are described as optional. This can be a function, component, mode, value or setting that the ARM partner can choose to include at implementation.

See the *Glossary* for a description of the term implementation defined.

In this book, unless the context indicates otherwise:

Processor   Refers to the Cortex-M7 processor, as supplied by ARM.

Device   Refers to an implemented device, supplied by an ARM partner, that incorporates a Cortex-M7 processor. In particular, your device refers to the particular implementation of the Cortex-M7 that you are using. Some features of your device depend on the implementation choices made by the ARM partner that made the device.

Product revision status

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

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

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

Intended audience

This book is written for application and system-level software developers, familiar with programming, who want to program a device that includes the Cortex-M7 processor.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to the Cortex-M7 processor and its features.

Chapter 2 *The Cortex-M7 Processor*

Read this for information about how to program the processor, the processor memory model, exception and fault handling, and power management.

Chapter 3 *The Cortex-M7 Instruction Set*

Read this for information about the processor instruction set.

Chapter 4 *Cortex-M7 Peripherals*

Read this for information about Cortex-M7 core peripherals.

Appendix A *Cortex-M7 Options*

Read this for information about the processor implementation and configuration options.

Appendix B *Revisions*

Read this for a list of the technical changes between released issues of this book.

Glossary Read this for definitions of terms used in this book.

Typographical conventions

The following table describes the typographical conventions:

Typographical conventions

Style	Purpose
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
<u>mono</u>space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This section lists publications by ARM and by third parties.

See *Infocenter*, http://infocenter.arm.com, for access to ARM documentation.

See *on ARM*, www.arm.com/cmsis, for embedded software development resources including the *Cortex® Microcontroller Software Interface Standard* (CMSIS).

ARM publications

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

*   *Cortex®-M7 Technical Reference Manual* (ARM DDI 0489).
*   *ARM®v7-M Architecture Reference Manual* (ARM DDI 0403).

Other publications

This guide only provides generic information for devices that implement the ARM Cortex-M7 processor. For information about your device see the documentation published by the device manufacturer.

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on content

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

*   The title.
*   The number, ARM DUI 0646A.
*   The page numbers to which your comments apply.
*   A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

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

Chapter 1
Introduction
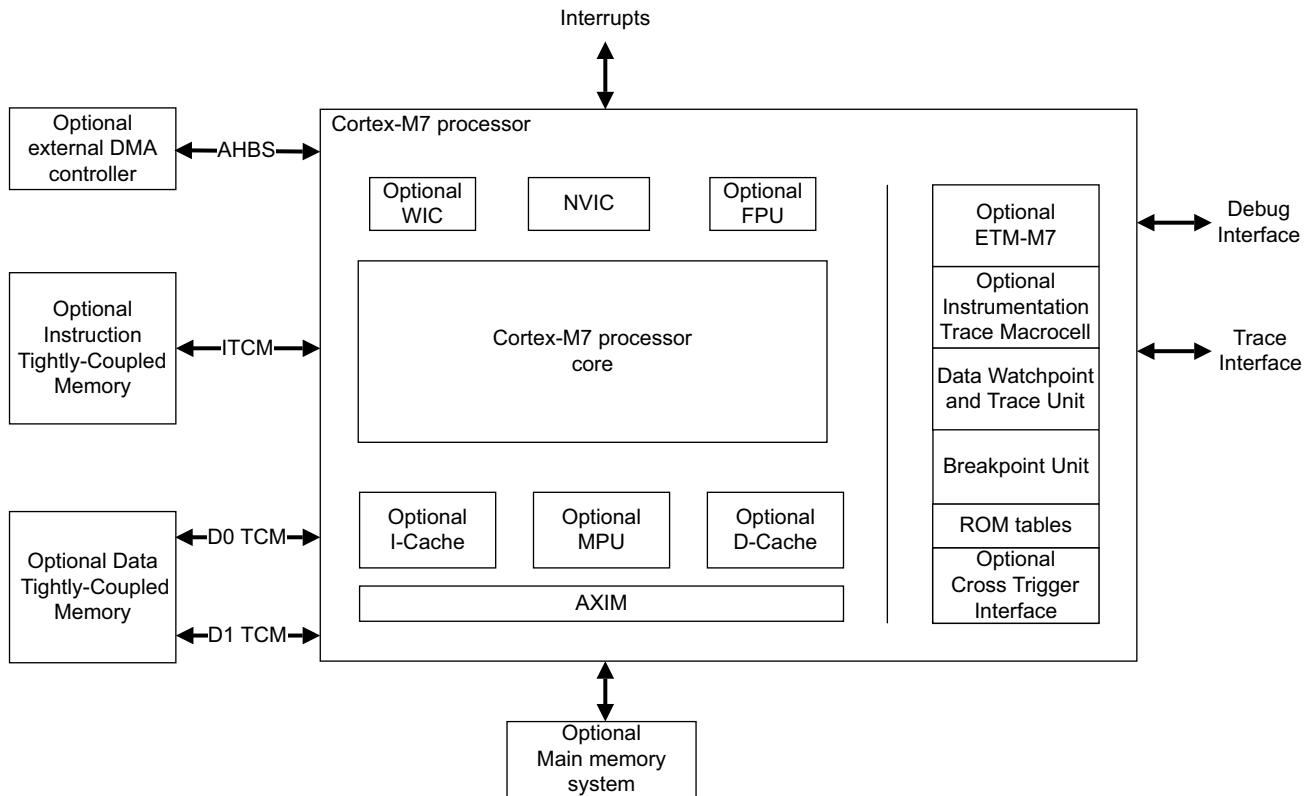
This chapter introduces the Cortex-M7 processor and its features. It contains the following section:

- *About the Cortex-M7 processor and core peripherals* on page 1-2.

1.1 About the Cortex-M7 processor and core peripherals

The Cortex-M7 processor is a high performance 32-bit processor designed for the microcontroller market. It offers significant benefits to developers, including:

- Outstanding processing performance combined with fast interrupt handling.
- Enhanced system debug with extensive breakpoint and trace capabilities.
- Efficient processor core, system and memories.
- Ultra-low power consumption with integrated sleep mode and an optional deep sleep mode.
- Platform security robustness, with optional integrated *Memory Protection Unit* (MPU).

Figure 1-1 Cortex-M7 processor implementation

The Cortex-M7 processor is built on a high-performance processor core, with a 6-stage pipeline Harvard architecture, making it ideal for demanding embedded applications. The in-order superscalar processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design, providing high-end processing hardware that includes a range of single-cycle and SIMD multiplication and multiply-with-accumulate capabilities, saturating arithmetic and dedicated hardware division and optional floating-point computation IEEE754-compliant options of either single-precision, or both single-precision and double-precision.

To facilitate the design of cost-sensitive devices, the Cortex-M7 processor implements tightly-coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex-M7 processor implements a version of the Thumb® instruction set based on Thumb-2 technology, ensuring high code density and reduced program memory requirements. The Cortex-M7 processor instruction set provides the exceptional performance expected of a modern 32-bit architecture, with the high code density of 8-bit and 16-bit microcontrollers.

The Cortex-M7 processor closely integrates a configurable *Nested Vectored Interrupt Controller* (NVIC), to deliver industry-leading interrupt performance. The NVIC includes a *Non-Maskable Interrupt* (NMI), and can provide up to 256 interrupt priority levels. The tight integration of the processor core and NVIC provides fast execution of *interrupt service routines* (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to suspend load-multiple and store-multiple operations. Interrupt handlers do not require wrapping in assembler code, removing any code overhead from the ISRs. A tail-chain optimization also significantly reduces the overhead when switching from one ISR to another.

To optimize low-power designs, the NVIC integrates with the sleep modes that includes an optional deep sleep function. This enables the entire device to be rapidly powered down while still retaining program state.

Reliability can be increased with optional automatic built-in fault detection and handling. With this option, the Cortex-M7 processor has *Memory Built-in Self Test* (MBIST) capability, and supports ECC (*Error Correcting Code*) on cache and TCM memories. This enables SEC-DED (*Single Error Correct, Double Error Detect*) for accesses to memory. The Cortex-M7 processor is dual-redundant, which means it can operate in lock-step. The MCU vendor determines the reliability features configuration and therefore this can differ across different devices and families.

To increase instruction throughput, the Cortex-M7 processor can execute certain pairs of instructions simultaneously. This is called dual issue.

1.1.1 System level interface

The Cortex-M7 processor provides multiple interfaces using AMBA® technology to provide high speed, low latency memory accesses. It supports unaligned data accesses.

The Cortex-M7 processor has an optional *Memory Protection Unit* (MPU) that provides fine grain memory control, enabling applications to utilize 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.

1.1.2 Integrated configurable debug

The Cortex-M7 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 and therefore this can differ across different devices and families.

For system trace the processor integrates an *Instrumentation Trace Macrocell* (ITM) together with data watchpoints and a profiling unit. To enable simple and cost-effective profiling of the system events these generate, a *Serial Wire Viewer* (SWV) can export a stream of software-generated messages, data trace, and profiling information through a single pin.

The optional CoreSight technology components, *Embedded Trace Macrocell* (ETM) and *Cross Trigger Interface* (CTI), deliver unrivalled instruction trace, and implementation-defined data trace, capture in an area far smaller than traditional trace units, enabling many low cost MCUs to implement full instruction trace for the first time.

The Breakpoint Unit can provide up to eight hardware breakpoint comparators that debuggers can use.

1.1.3 Cortex-M7 processor features and benefits summary

- Tight integration of system peripherals reduces area and development costs.
- Thumb instruction set combines high code density with 32-bit performance.
- Optional IEEE754-compliant single-precision, and implementation-defined double-precision, *Floating-Point Unit* (FPU).
- Power control optimization of system components.
- Integrated sleep modes for low power consumption.
- Fast code execution permits slower processor clock or increases sleep mode time.
- Hardware division and fast digital-signal-processing orientated multiply accumulate.
- Saturating arithmetic for signal processing.
- Deterministic, high-performance interrupt handling for time-critical applications.
- Optional MPU for safety-critical applications.
- Optional memory system features such as caches, *Tightly-Coupled Memory* (TCM) with DMA port, and a high performance AXI external memory interface.
- Extensive debug and trace capabilities:
  — Serial Wire Debug and Serial Wire Trace reduce the number of pins required for debugging, tracing, and code profiling.

1.1.4 Cortex-M7 processor core peripherals

The Cortex-M7 processor core peripherals are:

Nested Vectored Interrupt Controller

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

System Control Block

> The *System Control Block* (SCB) is the programmers model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.

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.

Integrated instruction and data caches (optional)

> The instruction and data caches provide fast access to frequently accessed data and instructions, providing support for increased average performance when using system based memory.

Memory Protection Unit (optional)

> The *Memory Protection Unit* (MPU) improves system reliability by defining the memory attributes for different memory regions. Depending on your implementation, it provides up to 8 or 16 different regions, and an optional predefined background region.

Floating-point unit (optional)

> The FPU provides IEEE754-compliant operations on 32-bit single-precision and 64-bit double-precision floating-point values.

Chapter 2
The Cortex-M7 Processor

This chapter describes how to program the Cortex-M7 processor. It contains the following sections:

2.1 Programmers model

This section describes the Cortex-M7 processor programmers model. In addition to the individual core register descriptions, it contains information about the processor modes and privilege levels for software execution and stacks.

2.1.1 Processor mode and privilege levels for software execution

The processor *modes* are:

Thread mode   Used to execute application software. The processor enters Thread mode when it comes out of reset.

Handler mode   Used to handle exceptions. The processor returns to Thread mode when it has finished all exception processing.

The *privilege levels* for software execution are:

Unprivileged   The software:

- Has limited access to the MSR and MRS instructions, and cannot use the CPS instruction.
- Cannot access the system timer, NVIC, or system control block.
- Might have restricted access to memory or peripherals.

Unprivileged software executes at the unprivileged level.

Privileged   The software can use all the instructions and has access to all resources.

Privileged software executes at the privileged level.

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged, see *CONTROL register* on page 2-9. In Handler mode, software execution is always privileged.

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

2.1.2 Stacks

The processor uses a full descending stack. This means 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. The processor implements two stacks, the *main stack* and the *process stack*, with a pointer for each held in independent registers, see *Stack Pointer* on page 2-4.

In Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack, see *CONTROL register* on page 2-9. In Handler mode, the processor always uses the main stack. The options for processor operations are:
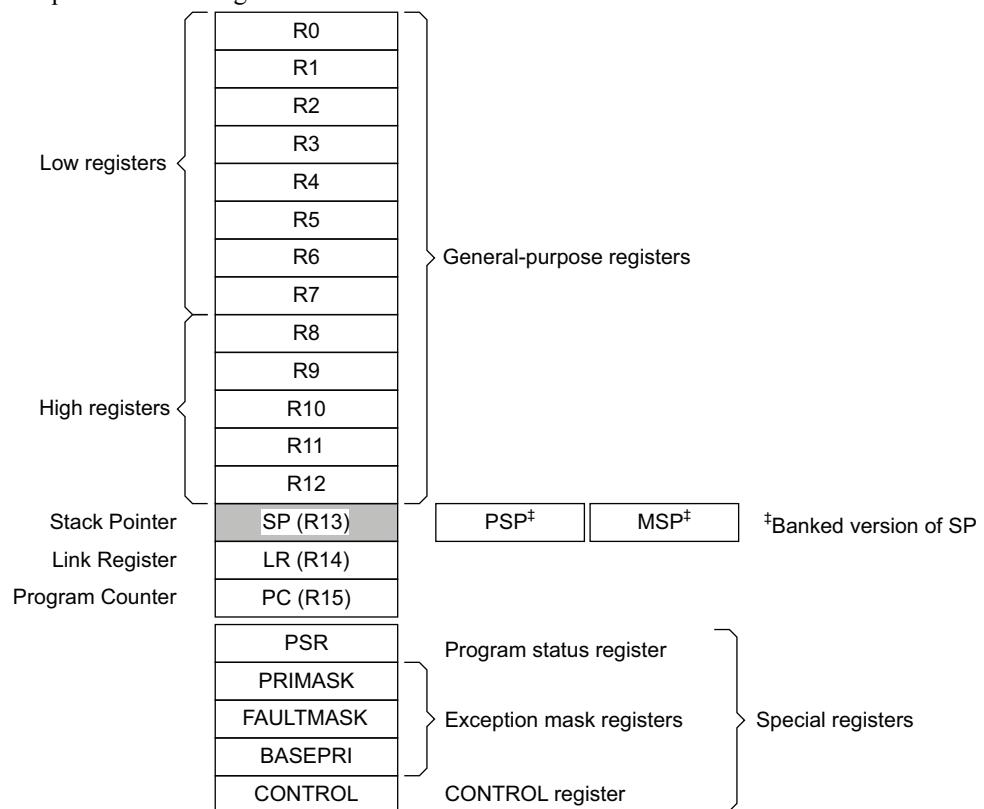
Table 2-1 Summary of processor mode, execution privilege level, and stack use options

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged[a]	Main stack or process stack[a]
Handler	Exception handlers	Always privileged	Main stack

    a. See *CONTROL register* on page 2-9.

2.1.3 Core registers

The processor core registers are:

	R0
	R1
	R2
	R3
Low registers	R4
	R5
	R6
	R7
	R8
	R9
High registers	R10
	R11
	R12

General-purpose registers

Stack Pointer — SP (R13) | PSP‡ | MSP‡ | ‡Banked version of SP
Link Register — LR (R14)
Program Counter — PC (R15)

PSR — Program status register
PRIMASK
FAULTMASK — Exception mask registers
BASEPRI
CONTROL — CONTROL register

Special registers

Table 2-2 Core register set summary

Name	Type[a]	Required privilege[b]	Reset value	Description
R0-R12	RW	Either	Unknown	*General-purpose registers* on page 2-4
MSP	RW	Either	See description	*Stack Pointer* on page 2-4
PSP	RW	Either	Unknown	*Stack Pointer* on page 2-4
LR	RW	Either	0xFFFFFFFF	*Link Register* on page 2-4
PC	RW	Either	See description	*Program Counter* on page 2-4
PSR	RW	Either	0x01000000[c]	*Program Status Register* on page 2-4
ASPR	RW	Either	Unknown	*Application Program Status Register* on page 2-5
IPSR	RO	Privileged	0x00000000	*Interrupt Program Status Register* on page 2-6
EPSR	RO	Privileged	0x01000000[c]	*Execution Program Status Register* on page 2-6
PRIMASK	RW	Privileged	0x00000000	*Priority Mask Register* on page 2-8
FAULTMASK	RW	Privileged	0x00000000	*Fault Mask Register* on page 2-8
BASEPRI	RW	Privileged	0x00000000	*Base Priority Mask Register* on page 2-8
CONTROL	RW	Privileged	0x00000000	*CONTROL register* on page 2-9

a. Describes access type during program execution in thread mode and Handler mode. Debug access can differ.

b. An entry of Either means privileged and unprivileged software can access the register.

c.   The EPSR reads as zero when executing an `MRS` instruction.

General-purpose registers

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

Stack Pointer

The *Stack Pointer* (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

0   *Main Stack Pointer* (MSP). This is the reset value.

1   *Process Stack Pointer* (PSP).

On reset, the processor loads the MSP with the value from the implementation-defined address `0x00000000`.

Link Register

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

Program Counter

The *Program Counter* (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector, that is at the implementation-defined address `0x00000004`. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program Status Register

The *Program Status Register* (PSR) combines:
*   *Application Program Status Register* (APSR).
*   *Interrupt Program Status Register* (IPSR).
*   *Execution Program Status Register* (EPSR).

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

	31 30 29 28 27	26 25 24 23 20 19 16	15 10 9 8 0
APSR	N Z C V Q	Reserved	GE[3:0]	Reserved
IPSR	Reserved		ISR_NUMBER
EPSR	Reserved	ICI/IT	T	Reserved	ICI/IT	Reserved

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

The PSR combinations and attributes are:

Table 2-3 PSR register combinations

Register	Type	Combination
PSR	RW[a, b]	APSR, EPSR, and IPSR
IEPSR	RO[b]	EPSR and IPSR
IAPSR	RW[a]	APSR and IPSR
EAPSR	RW[b]	APSR and EPSR

a. The processor ignores writes to the IPSR bits.

b. Reads of the EPSR bits return zero, and the processor ignores writes to the these bits

See the instruction descriptions *MRS* on page 3-178 and *MSR* on page 3-179 for more information about how to access the program status registers.

Application Program Status Register

The APSR contains the current state of the condition flags from previous instruction executions. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-4 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 *SEL* on page 3-77 for more information.
[15:0]	-	Reserved.

Interrupt Program Status Register

The IPSR contains the exception type number of the current *Interrupt Service Routine* (ISR). See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-5 IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved.
[8:0]	ISR_NUMBER	This is the number of the current exception:
		0 Thread mode.
		1 Reserved.
		2 NMI.
		3 HardFault.
		4 MemManage.
		5 BusFault.
		6 UsageFault.
		7-10 Reserved.
		11 SVCall.
		12 Reserved for Debug.
		13 Reserved.
		14 PendSV.
		15 SysTick.
		16 IRQ0.
		. .
		. .
		. .
		255 IRQ239[a].
		See *Exception types* on page 2-19 for more information.

a. This is the maximum number of exceptions. The actual maximum number of exceptions available is implementation defined.

Execution Program Status Register

The EPSR contains the Thumb state bit, and the execution state bits for either the:

• *If-Then* (IT) instruction.

• *Interruptible-Continuable Instruction* (ICI) field for an interrupted load multiple or store multiple instruction.

See the register summary in Table 2-2 on page 2-3 for the EPSR attributes. The bit assignments are:

Table 2-6 EPSR bit assignments

Bits	Name	Function
[31:27]	-	Reserved
[26:25], [15:10]	ICI	Interruptible-continuable instruction bits, see *Interruptible-continuable instructions* on page 2-7
[26:25], [15:10]	IT	Indicates the execution state bits of the IT instruction, see *IT* on page 3-131

Bits	Name	Function
[24]	T	Thumb state bit, see *Thumb state*
[23:16]	-	Reserved
[9:0]	-	Reserved

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

Interruptible-continuable instructions

When an interrupt occurs during the execution of an `LDM`, `STM`, `PUSH` or `POP` instruction, and when an FPU is implemented an `VLDM`, `VSTM`, `VPUSH`, or `VPOP` instruction, the processor:
- Stops the load multiple or store multiple instruction operation temporarily.
- Stores the next register operand in the multiple operation to EPSR bits[15:12].

After servicing the interrupt, the processor:
- Returns to the register pointed to by bits[15:12].
- Resumes execution of the multiple load or store instruction.

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

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. See *IT* on page 3-131 for more information.

Thumb state

The Cortex-M7 processor only supports execution of instructions in Thumb state. The following can clear the T bit to 0:
- Instructions `BLX`, `BX` 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 *Lockup* on page 2-29 for more information.
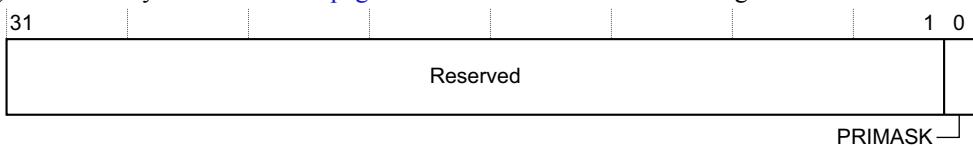
Exception mask registers

The exception mask registers disable the handling of exceptions by the processor. Disable exceptions where they might impact on 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 or FAULTMASK. See *MRS* on page 3-178, *MSR* on page 3-179, and *CPS* on page 3-174 for more information.

Priority Mask Register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

```
31                                                                    1  0
┌──────────────────────────────────────────────────────────────────┬──┐
│                           Reserved                                 │  │
└──────────────────────────────────────────────────────────────────┴──┘
                                                            PRIMASK ──┘
```
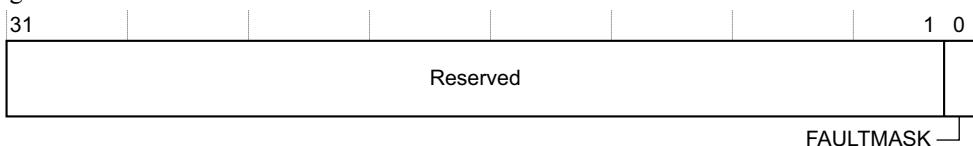
Table 2-7 PRIMASK register bit assignments

Bits	Name	Function	
[31:1]	-	Reserved.	
[0]	PRIMASK	**0**	No effect.
		1	Prevents the activation of all exceptions with configurable priority.

Fault Mask Register

The FAULTMASK register prevents activation of all exceptions except for *Non-Maskable Interrupt* (NMI). See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

```
31                                                                    1  0
┌──────────────────────────────────────────────────────────────────┬──┐
│                           Reserved                                 │  │
└──────────────────────────────────────────────────────────────────┴──┘
                                                          FAULTMASK ──┘
```

Table 2-8 FAULTMASK register bit assignments

Bits	Name	Function	
[31:1]	-	Reserved.	
[0]	FAULTMASK	**0**	No effect.
		1	Prevents the activation of all exceptions except for NMI.

The processor clears the FAULTMASK bit to 0 on exit from any exception handler except the NMI handler.

Base Priority Mask Register

The BASEPRI register defines the minimum priority for exception processing. When BASEPRI is set to a nonzero value, it prevents the activation of all exceptions with the same or lower priority level as the BASEPRI value. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

```
31                                              8  7              0
┌───────────────────────────────────────────────┬────────────────┐
│                   Reserved                     │    BASEPRI     │
└───────────────────────────────────────────────┴────────────────┘
```
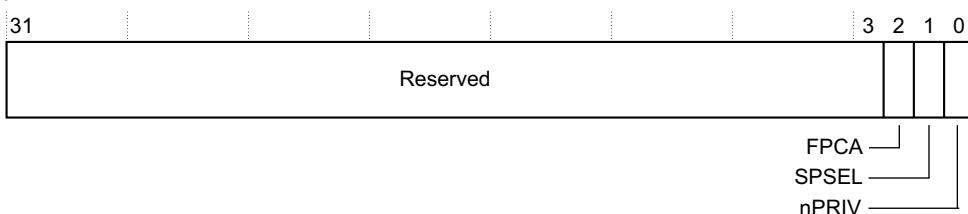
Table 2-9 BASEPRI register bit assignments

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	BASEPRI[a]	Priority mask bits: 0x00 No effect. Nonzero Defines the base priority for exception processing. The processor does not process any exception with a priority value greater than or equal to BASEPRI.

a. This field is similar to the priority fields in the interrupt priority registers. Only bits[7:M] of this field are implemented, and bits[M-x:0] read as zero and ignore writes. The values of M and x are implementation defined. See *Interrupt Priority Registers* on page 4-7 for more information. Remember that higher priority field values correspond to lower exception priorities.

CONTROL register

The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode, and if implemented, indicates whether the FPU state is active. See the register summary in Table 2-2 on page 2-3 for its attributes. The bit assignments are:

Table 2-10 CONTROL register bit assignments

Bits	Name	Function
[31:3]	-	Reserved.
[2]	FPCA	When floating-point is implemented this bit indicates whether floating-point context is currently 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.
[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-M7 processor updates this bit automatically on exception return.
[0]	nPRIV	Defines the Thread mode privilege level: **0** Privileged. **1** Unprivileged.

Handler mode always uses the MSP, so the processor ignores explicit writes to the active stack pointer bit of the CONTROL register when in Handler mode. The exception entry and return mechanisms automatically update the CONTROL register based on the EXC_RETURN value, see Table 2-15 on page 2-26.

In an OS environment, the vendor 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 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, see *MSR* on page 3-179.

- Perform an exception return to Thread mode with the appropriate EXC_RETURN value, see Table 2-15 on page 2-26.

————— **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. See *ISB* on page 3-177.

2.1.4 Exceptions and interrupts

The Cortex-M7 processor supports interrupts and system exceptions. The processor and the NVIC prioritize and handle all exceptions. An exception changes the normal flow of software control. The processor uses Handler mode to handle all exceptions except for reset. See *Exception entry* on page 2-24 and *Exception return* on page 2-26 for more information.

The NVIC registers control interrupt handling. See *Nested Vectored Interrupt Controller* on page 4-3 for more information.

2.1.5 Data types

The processor:

- Supports the following data types:
    — 32-bit words.
    — 16-bit halfwords.
    — 8-bit bytes.
    — 32-bit single-precision floating point numbers if the FPU is implemented.
    — 64-bit double-precision floating point numbers if the FPU is implemented.

- Manages all data memory accesses as either little-endian or big-endian depending on how your implementation is defined. Instruction memory and *Private Peripheral Bus* (PPB) accesses are always performed as little-endian. See *Memory regions, types and attributes* on page 2-12 for more information.

2.1.6 The Cortex Microcontroller Software Interface Standard

For a Cortex-M7 microcontroller system, the *Cortex Microcontroller Software Interface Standard* (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-M7 processor.

CMSIS simplifies software development by enabling the reuse of template code and the combination of CMSIS-compliant software components from various middleware vendors. Software vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals.

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

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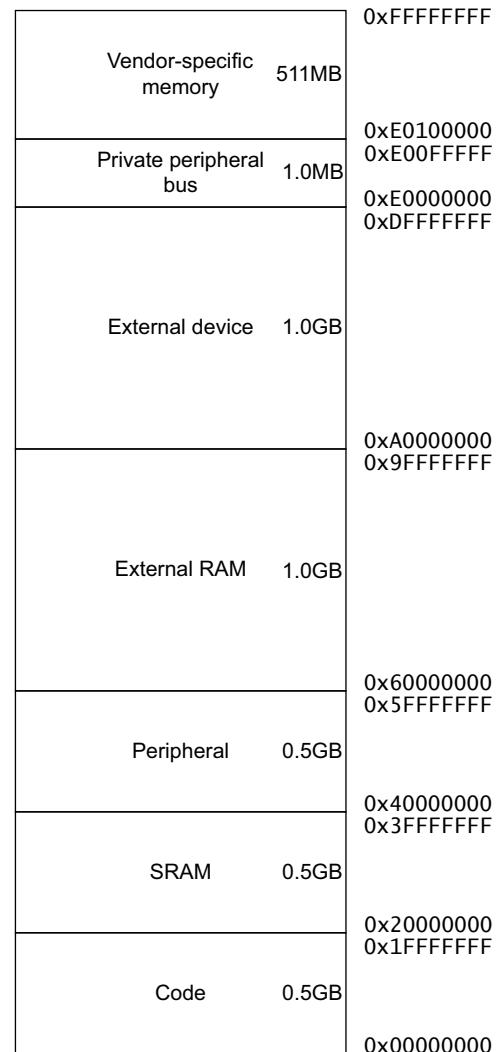
This document uses the register short names defined by the CMSIS. In a few cases these differ from the architectural short names that might be used in other documents.

The following sections give more information about the CMSIS:

2.2   Memory model

This section describes the processor memory map and the behavior of memory accesses. The processor has a fixed default memory map that provides up to 4GB of addressable memory. The memory map is:

Vendor-specific memory	511MB	0xFFFFFFFF
		0xE0100000
Private peripheral bus	1.0MB	0xE00FFFFF
		0xE0000000
External device	1.0GB	0xDFFFFFFF
		0xA0000000
External RAM	1.0GB	0x9FFFFFFF
		0x60000000
Peripheral	0.5GB	0x5FFFFFFF
		0x40000000
SRAM	0.5GB	0x3FFFFFFF
		0x20000000
Code	0.5GB	0x1FFFFFFF
		0x00000000

The processor reserves regions of the *Private peripheral bus* (PPB) address range for core peripheral registers, see *About the Cortex-M7 peripherals* on page 4-2.

2.2.1   Memory regions, types and attributes

The memory map and programming the optional MPU splits the memory map into regions. Each region has a defined memory type, and some regions have additional memory attributes. The memory type and attributes determine the behavior of accesses to the region.

The memory types are:

Normal   The processor can re-order transactions for efficiency, or perform speculative reads.

Device and Strongly-ordered

The processor preserves transaction order relative to other transactions to Device or Strongly-ordered memory.

The different ordering requirements for Device and Strongly-ordered memory mean that the external memory system can buffer a write to Device memory, but must not buffer a write to Strongly-ordered memory.

The additional memory attributes include:

Shareable
For a shareable memory region that is implemented, the memory system provides data synchronization between bus masters in a system with multiple bus masters, for example, a processor with a DMA controller.

Strongly-ordered memory is always shareable.

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

Execute Never (XN)

Means the processor prevents instruction accesses. A fault exception is generated only on execution of an instruction executed from an XN region.

2.2.2 Memory system ordering of memory accesses

For most memory accesses caused by explicit memory access instructions, the memory system does not guarantee that the order in which the accesses complete matches the program order of the instructions, providing this does not affect the behavior of the instruction sequence. Normally, if correct program execution depends on two memory accesses completing in program order, software must insert a memory barrier instruction between the memory access instructions, see *Software ordering of memory accesses* on page 2-15.

However, the memory system does guarantee some ordering of accesses to Device and Strongly-ordered memory. For two memory access instructions A1 and A2, if A1 occurs before A2 in program order, the ordering of the memory accesses caused by two instructions is:

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

Where:

-
Means that the memory system does not guarantee the ordering of the accesses.

<
Means that accesses are observed in program order, that is, A1 is always observed before A2.

2.2.3 Behavior of memory accesses

The behavior of accesses to each region in the memory map is:

Table 2-11 Memory access behavior

Address range	Memory region	Memory type[a]	XN[a]	Description
0x00000000-0x1FFFFFFF	Code	Normal	-	Executable region for program code. You can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal	-	Executable region for data. You can also put code here.
0x40000000-0x5FFFFFFF	Peripheral	Device	XN	Peripheral address space.
0x60000000-0x9FFFFFFF	External RAM	Normal	-	Executable region for data. You can also put code here.
0xA0000000-0xDFFFFFFF	External device	Device	XN	External Device memory.
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	XN	This region includes the NVIC, System timer, and system control block.
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device	XN	Accesses to this region are to vendor-specific peripherals.

a. See *Memory regions, types and attributes* on page 2-12 for more information.

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

The optional MPU can override the default memory access behavior described in this section. For more information, see *Optional Memory Protection Unit* on page 4-43.

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, as Table 2-12 shows:

Table 2-12 Memory region shareability and cache policies

Address range	Memory region	Memory type	Shareability	Cache policy
0x00000000-0x1FFFFFFF	Code	Normal[a]	Non-shareable[a]	WT[b]
0x20000000-0x3FFFFFFF	SRAM	Normal[a]	Non-shareable[a]	WBWA[b]
0x40000000-0x5FFFFFFF	Peripheral	Device[a]	Non-shareable[a]	-
0x60000000-0x7FFFFFFF	External RAM	Normal[a]	Non-shareable[a]	WBWA[b]
0x80000000-0x9FFFFFFF				WT[b]
0xA0000000-0xBFFFFFFF	External device	Device[a]	Shareable[a]	-
0xC0000000-0xDFFFFFFF			Non-shareable[a]	
0xE0000000-0xE00FFFFF	Private Peripheral Bus	Strongly-ordered[a]	Shareable[a]	-
0xE0100000-0xFFFFFFFF	Vendor-specific device	Device[a]	Non-shareable[a]	-

a. See *Memory regions, types and attributes* on page 2-12 for more information.

b. WT = Write through, no write allocate. WBWA = Write back, write allocate.

Instruction prefetch and branch prediction

The Cortex-M7 processor:

- Prefetches instructions ahead of execution.
- Speculatively prefetches from branch target addresses.

2.2.4 Software ordering of memory accesses

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

- The processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence.

- The processor has multiple bus interfaces.

- Memory or devices in the memory map have different wait states.

- Some memory accesses are buffered or speculative.

Memory system ordering of memory accesses on page 2-13 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. 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. See *DMB* on page 3-175.

DSB   The *Data Synchronization Barrier* (DSB) instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See *DSB* on page 3-176.

ISB   The *Instruction Synchronization Barrier* (ISB) ensures that the effect of all completed memory transactions is recognizable by subsequent instructions. See *ISB* on page 3-177.

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.

A detailed explanation of all the cases in which a Barrier might be required is included in *ARM® Cortex®-M Programming Guide to Memory Barrier Instructions Application Note 321*.
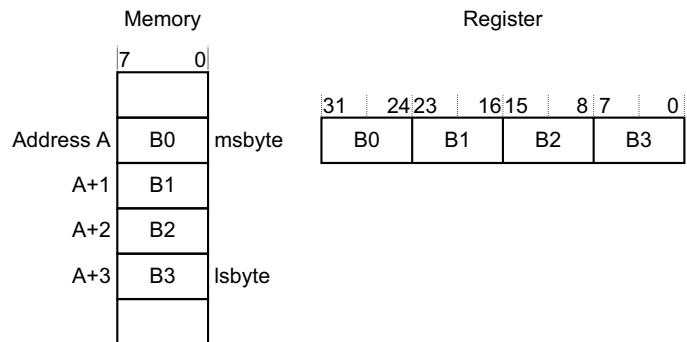
2.2.5 Memory endianness

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. The memory endianness used is one of the following depending on your implementation:

- *Byte-invariant big-endian format* on page 2-16.
- *Little-endian format* on page 2-16.
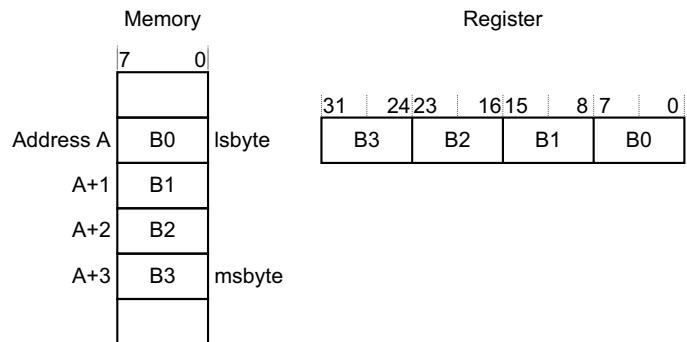
Byte-invariant big-endian format

In byte-invariant big-endian format, the processor stores the most significant byte of a word at the lowest-numbered byte, and the least significant byte at the highest-numbered byte. For example:

```
             Memory                              Register
          7        0
                                   31    24 23   16 15    8 7     0
Address A    B0     msbyte          B0      B1      B2      B3
   A+1       B1
   A+2       B2
   A+3       B3     lsbyte
```

Little-endian format

In little-endian format, the processor stores the least significant byte of a word at the lowest-numbered byte, and the most significant byte at the highest-numbered byte. For example:

```
             Memory                              Register
          7        0
                                   31    24 23   16 15    8 7     0
Address A    B0     lsbyte          B3      B2      B1      B0
   A+1       B1
   A+2       B2
   A+3       B3     msbyte
```

2.2.6 Synchronization primitives

The instruction set support for the Cortex-M7 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 perform a guaranteed read-modify-write memory update sequence, or for a semaphore mechanism.

A pair of synchronization primitives comprises:

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 succeeds,

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

The pairs of Load-Exclusive and Store-Exclusive instructions are:
- The word instructions LDREX and STREX.
- The halfword instructions LDREXH and STREXH.
- The byte instructions LDREXB and STREXB.

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

To perform an exclusive read-modify-write of a memory location, 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.

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.

The Cortex-M7 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 and the address is in a shared region of memory, the system also globally tags the memory locations addressed by exclusive accesses by each processor.

The processor removes its exclusive access tag if:

- It executes a CLREX instruction.

- It executes a STREX instruction, regardless of whether the write succeeds.

- An exception occurs. This means the processor can resolve semaphore conflicts between different threads.

In a multiprocessor implementation:

- Executing a CLREX instruction removes only the local exclusive access tag for the processor.

- Executing a STREX instruction, or an exception, removes the local exclusive access tags for the processor.

- Executing a STREX instruction to a shared memory region can also remove the global exclusive access tags for the processor in the system.

For more information about the synchronization primitive instructions, see *LDREX and STREX on page 3-39* and *CLREX on page 3-41*.

2.2.7 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 2-13 CMSIS functions for exclusive access instructions

Instruction	CMSIS function
LDREX	uint32_t __LDREXW (uint32_t *addr)
LDREXH	uint16_t __LDREXH (uint16_t *addr)
LDREXB	uint8_t __LDREXB (uint8_t *addr)
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXH	uint32_t __STREXH (uint16_t value, uint16_t *addr)
STREXB	uint32_t __STREXB (uint8_t value, uint8_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
```

2.2.8 Dynamic read allocate mode

When a memory region is marked as Write-Back Write-Allocate, it normally allocates a cache line on either a read miss or a write miss. However, there are some situations where allocating on writes is undesirable, such as executing the C standard library memset() function to clear a large block of memory to a known value. Writing large blocks of data like this can pollute the cache with unnecessary data. It can also waste power and performance if a linefill must be performed only to discard the linefill data because the entire line was subsequently written by the memset().

To prevent this, the Cortex-M7 processor *Bus Interface Unit* (BIU) includes logic to detect when a full cache line has been written by the core before the linefill has completed. If this situation is detected on three consecutive linefills, it switches into dynamic read allocate mode. When in dynamic read allocate mode, loads behave as normal and can still cause linefills, and writes still lookup in the cache but, if they miss, they write out to external memory rather than starting a linefill.

The BIU continues in dynamic read allocate mode until it detects either a cacheable write burst to external memory that is not a full cache line, or there is a load to the same line as is currently being written to external memory.

Dynamic read allocate mode can be disabled by setting the ACTLR.DISRAMODE to 1. See *Auxiliary Control Register* on page 4-11.

2.3 Exception model

This section describes the exception model. It describes:

- *Exception states*.
- *Exception types*.
- *Exception handlers* on page 2-21.
- *Vector table* on page 2-21.
- *Exception priorities* on page 2-23.
- *Interrupt priority grouping* on page 2-23.
- *Exception entry and return* on page 2-23.

2.3.1 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 that 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.

2.3.2 Exception types

The exception types are:

Reset   Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. It is permanently enabled and has a fixed priority of -3. When 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 prevented from activation by any other exception.
- 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 related fault. The fixed memory protection constraints, or the MPU if implemented, determines this fault, 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 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 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.

PendSV   PendSV is an interrupt-driven 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 system tick.

Interrupt (IRQ)   A 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 2-14 Properties of the different exception types

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address or offset[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable[c]	0x00000010	Synchronous
5	-11	BusFault	Configurable[c]	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable[c]	0x00000018	Synchronous
7-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[c]	0x0000002C	Synchronous

Table 2-14 Properties of the different exception types (continued)

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address or offset[b]	Activation
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[c]	0x00000038	Asynchronous
15	-1	SysTick	Configurable[c]	0x0000003C	Asynchronous
16 and above[d]	0 and above[e]	Interrupt (IRQ)	Configurable[f]	0x00000040 and above[g]	Asynchronous

a. To simplify the software layer, the CMSIS only uses IRQ numbers and therefore uses negative values for exceptions other than interrupts. The IPSR returns the Exception number, see *Interrupt Program Status Register* on page 2-6.

b. See *Vector table* for more information.

c. See *System Handler Priority Registers* on page 4-22.

d. The exception number range is implementation defined.

e. The IRQ number range is implementation defined.

f. See *Interrupt Priority Registers* on page 4-7.

g. Increasing in steps of 4. The vector address or offset range is implementation defined.

For an asynchronous exception, other than reset, the processor can execute another instruction between when the exception is triggered and when the processor enters the exception handler.

Privileged software can disable the exceptions that Table 2-14 on page 2-20 shows as having configurable priority, see:

- *System Handler Control and State Register* on page 4-24.
- *Interrupt Clear-enable Registers* on page 4-5.

For more information about HardFaults, MemManage faults, BusFaults, and UsageFaults, see *Fault handling* on page 2-27.

2.3.3   Exception handlers

The processor handles exceptions using:

Interrupt Service Routines (ISRs)

Interrupts IRQ0 to IRQ239 is the maximum range of exceptions that can be handled by ISRs. The actual number of exceptions available is implementation defined.

Fault handlers   HardFault, MemManage fault, UsageFault, and BusFault are fault exceptions handled by the fault handlers.

System handlers   NMI, PendSV, SVCall SysTick, and the fault exceptions are all system exceptions that are handled by system handlers.

2.3.4   Vector table

The vector table contains the reset value of the stack pointer, and the start addresses, also called exception vectors, for all exception handlers. Figure 2-1 on page 2-22 shows the order of the exception vectors in the vector table. The least-significant bit of each vector must be 1, indicating that the exception handler is Thumb code, see *Thumb state* on page 2-7.

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
. . .	. . .	. . .	. . .
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5		SVCall
10		0x002C	
9			Reserved
8			
7			
6	-10		Usage fault
5	-11	0x0018	Bus fault
4	-12	0x0014	Memory management fault
3	-13	0x0010	Hard fault
2	-14	0x000C	NMI
1		0x0008	Reset
		0x0004	Initial SP value
		0x0000	

Figure 2-1 Vector table

Note

Figure 2-1 shows the maximum range of values for the exception number, IRQ number, offset and vector. The actual range of values available is implementation defined.

On system reset, the vector table is at the address configured at implementation, typically `0x00000000`. Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range `0x00000000-0xFFFFFF80`. The silicon vendor must configure the top range value, that is dependent 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, see *Vector Table Offset Register* on page 4-17.

ARM recommends that you locate the vector table in either the CODE, SRAM, External RAM, or External Device areas of the system memory map, see *Memory model* on page 2-12. Using the Peripheral, Private peripheral bus, or Vendor-specific memory areas can lead to unpredictable behavior in some systems. This is because the processor uses a different interfaces for load/store instructions and vector fetch in these memory areas. If the vector table is located in a region of memory that is cacheable, you must treat any load or store to the vector as self-modifying code and use cache maintenance instructions to synchronize the update to the data and instruction caches, see *Cache maintenance design hints and tips* on page 4-64.

2.3.5 Exception priorities

As Table 2-14 on page 2-20 shows, all exceptions have an associated priority, with:
- A lower priority value indicating a higher priority.
- Configurable 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:
- *System Handler Priority Registers* on page 4-22.
- *Interrupt Priority Registers* on page 4-7.

——— **Note** ———

The maximum range of configurable priority values is 0-255. The actual range of priority values available is implementation defined. This means that the Reset, HardFault, and NMI exceptions, with fixed negative priority values, always have higher priority than any other exception.

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

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

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

2.3.6 Interrupt priority grouping

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

Only the group priority determines preemption 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 preempt 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.

For information about splitting the interrupt priority fields into group priority and subpriority, see *Application Interrupt and Reset Control Register* on page 4-17.

2.3.7 Exception entry and return

Descriptions of exception handling use the following terms:

Preemption   When the processor is executing an exception handler, an exception can preempt the exception handler if its priority is higher than the priority of the exception being handled. See *Interrupt priority grouping* for more information about preemption by an interrupt.

When one exception preempts another, the exceptions are called nested exceptions. See *Exception entry* more information.

Return   This occurs when the exception handler is completed, and:

- There is no pending exception with sufficient priority to be serviced.

- The completed exception handler was not handling a late-arriving exception.

The processor pops the stack and restores the processor state to the state it had before the interrupt occurred. See *Exception return* on page 2-26 for more information.

Tail-chaining   This mechanism speeds up exception servicing. On completion of an exception handler, if there is a pending exception that meets the requirements for exception entry, the stack pop is skipped and control transfers to the new exception handler.

Late-arriving   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 is not affected by late arrival because the state saved is the same for both exceptions. Therefore the state saving continues uninterrupted. The processor can accept a late arriving exception until the first instruction of the exception handler of the original exception enters the execute stage of the processor. On return from the exception handler of the late-arriving exception, the normal tail-chaining rules apply.

Exception entry

Exception entry occurs when there is a pending exception with sufficient priority and either:

- The processor is in Thread mode.

- 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 the exception has more priority than any limits set by the mask registers, see *Exception mask registers* on page 2-7. An exception with less 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 eight data words is referred as the *stack frame*.

When using floating-point routines, the Cortex-M7 processor automatically stacks the architected floating-point state on exception entry. Figure 2-2 on page 2-25 shows the Cortex-M7 processor stack frame layout when floating-point state is preserved on the stack as the result of an interrupt or an exception.

───── **Note** ─────

Where stack space for floating-point state is not allocated, the stack frame is the same as that of ARMv7-M implementations without an FPU. Figure 2-2 on page 2-25 shows this stack frame also.
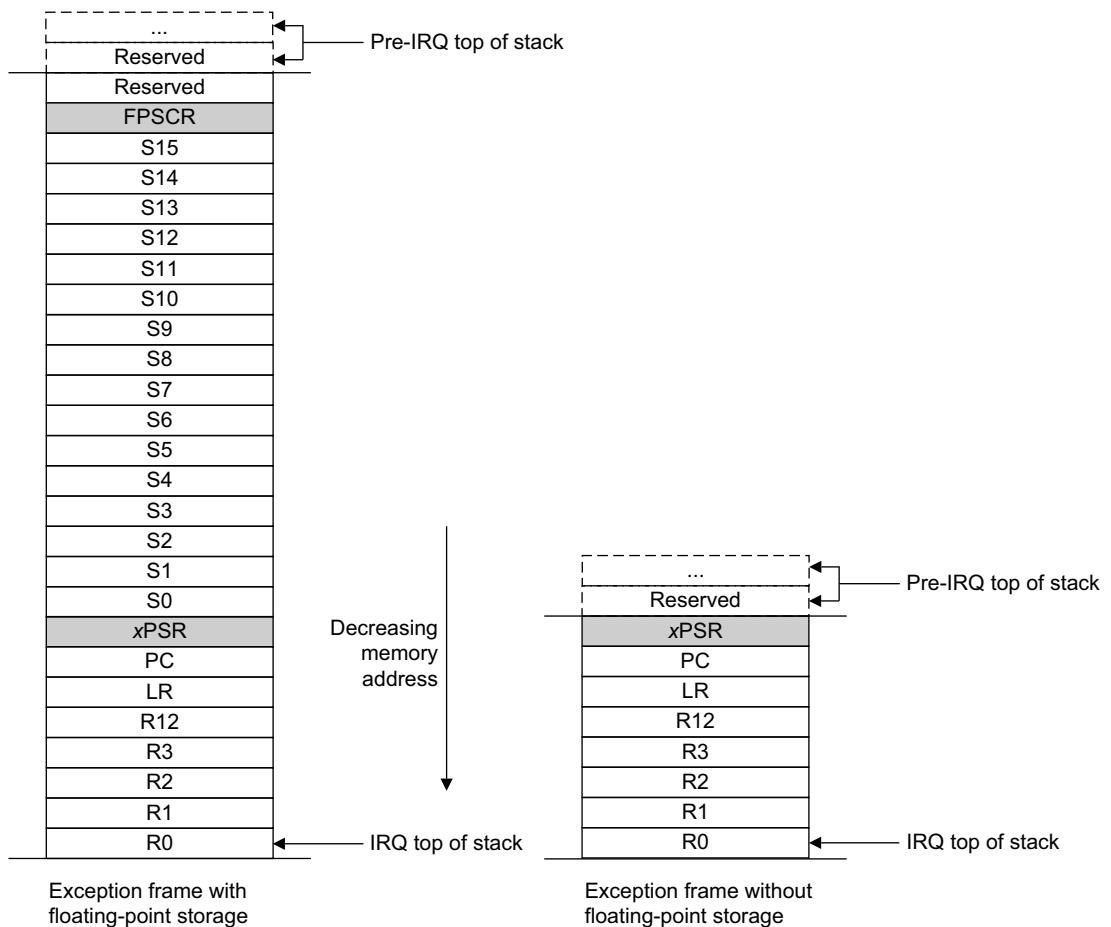
Figure 2-2 Exception stack frame

Immediately after stacking, the stack pointer indicates the lowest address in the stack frame. The alignment of the stack frame is controlled using the STKALIGN bit of the *Configuration Control Register* (CCR).

───── **Note** ─────

In the Cortex-M7 processor CCR.STKALIGN is read-only and has a value of 1. This means the stack address is always 8-byte aligned.

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 indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred.

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

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

Exception return

Exception return occurs when the processor is in Handler mode and executes one of the following instructions to load the EXC_RETURN value into the PC:

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

EXC_RETURN is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. The lowest five bits of this value provide information on the return stack and processor mode. Table 2-15 shows the EXC_RETURN values with a description of the exception return behavior.

All EXC_RETURN values have bits[31:5] set to one. When this value is loaded into the PC it indicates to the processor that the exception is complete, and the processor initiates the appropriate exception return sequence.

Table 2-15 Exception return behavior

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state[a] from MSP and execution uses MSP after return
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return

    a. Floating-point state is only possible if the optional FPU is implemented.

2.4   Fault handling

Faults are a subset of the exceptions, see *Exception model* on page 2-19. 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).

- If your device contains an MPU, a privilege violation or an attempt to access an unmanaged region causing an MPU fault.

2.4.1   Fault types

Table 2-16 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. See *Configuration and Control Register* on page 4-20 for more information about the fault status registers.

Table 2-16 Faults

Fault	Handler	Bit name	Fault status register
Bus error on a vector read	HardFault	VECTTBL	*HardFault Status Register* on page 4-31
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	-
On instruction access		IACCVIOL[a]	*MemManage Fault Status Register* on page 4-26
On data access[b]		DACCVIOL	
During exception stacking[b]		MSTKERR	
During exception unstacking[b]		MUNSKERR	
During lazy floating-point state preservation[c]		MLSPERR	
Bus error:	BusFault	-	-
During exception stacking		STKERR	*BusFault Status Register* on page 4-27
During exception unstacking		UNSTKERR	
During instruction prefetch		IBUSERR	
During lazy floating-point state preservation[d]		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	

Table 2-16 Faults (continued)

Fault	Handler	Bit name	Fault status register
Attempt to access a coprocessor	UsageFault	NOCP	*UsageFault Status Register* on page 4-29
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state[e]		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Divide By 0		DIVBYZERO	

a. Occurs on an access to an XN region even if the processor does not include an MPU or the MPU is disabled.

b. Occurs only if an MPU is implemented and enabled.

c. Occurs only if an MPU and FPU are implemented and enabled.

d. Occurs only if an FPU is implemented and enabled.

e. Attempting to use an instruction set other than the Thumb instruction set or returns to a non load/store-multiple instruction with ICI continuation.

2.4.2 Fault escalation and hard faults

All fault exceptions except for HardFault have configurable exception priority, see *System Handler Priority Registers* on page 4-22. Software can disable execution of the handlers for these faults, see *System Handler Control and State Register* on page 4-24.

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. as described in *Exception model* on page 2-19.

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 because it must have the same priority as the current 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.

——— **Note** ———

Only Reset and NMI can preempt the fixed priority HardFault. A HardFault can preempt any exception other than Reset, NMI, or another HardFault.

2.4.3 Synchronous and Asynchronous bus faults

In the Cortex-M7 processor all bus faults triggered by:

- Processor load operations are synchronous.

- Processor store operations are asynchronous, including stores to Device and Strongly-ordered regions.

- Debugger load or store accesses are synchronous, and are visible to the debugger interface only.

When an asynchronous bus fault is triggered, the BusFault exception is pended. If the BusFault handler is not enabled, the HardFault exception is pended instead. The HardFault caused by the asynchronous BusFault never escalates into lockup.

If an IRQ is triggered after the write, the write buffer might not drain before the ISR is executed. Therefore an asynchronous BusFault can occur across context boundaries.

A synchronous BusFault can escalate into lockup if it occurs inside a NMI or HardFault handler.

Cache maintenance operations can also trigger a bus fault. See *Faults handling considerations on page 4-64* for more information.

2.4.4 Fault status registers and fault address registers

The fault status registers indicate the cause of a fault. For synchronous BusFaults and MemManage faults, the fault address register indicates the address accessed by the operation that caused the fault, as shown in Table 2-17.

Table 2-17 Fault status and fault address registers

Handler	Status register name	Address register name	Register description
HardFault	HFSR	-	*HardFault Status Register* on page 4-31
MemManage	MMFSR	MMFAR	*MemManage Fault Status Register* on page 4-26 *MemManage Fault Address Register* on page 4-31
BusFault	BFSR	BFAR	*BusFault Status Register* on page 4-27 *BusFault Address Register* on page 4-32
UsageFault	UFSR	-	*UsageFault Status Register* on page 4-29

2.4.5 Lockup

The processor enters a lockup state if a fault occurs when executing the NMI or HardFault handlers. When the processor is in lockup state it does not execute any instructions. The processor remains in lockup state until either:
- It is reset.
- An NMI occurs.
- It is halted by a debugger.

——— **Note** ———

If lockup state occurs from the NMI handler a subsequent NMI does not cause the processor to leave lockup state.

2.5 Power management

The Cortex-M7 processor sleep modes reduce power consumption. The sleep modes your device implements is implementation defined. The modes implemented can be either one or both of the following:

- Sleep mode stops the processor clock.

- Deep sleep mode stops the system clock and switches off the PLL and flash memory.

If your device implements two sleep modes providing different levels of power saving, the SLEEPDEEP bit in the System Control Register (SCR) selects which sleep mode is used, see *System Control Register* on page 4-20. For more information about the behavior of the sleep modes see the documentation supplied by your device vendor.

This section describes the mechanisms for entering sleep mode, and the conditions for waking up from sleep mode.

2.5.1 Entering sleep mode

This section describes the mechanisms software can use to put the processor into sleep mode.

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

Wait for interrupt

The *wait for interrupt* instruction, WFI, causes immediate entry to sleep mode unless the wakeup condition is true, see *Wakeup from WFI or sleep-on-exit* on page 2-31. When the processor executes a WFI instruction it stops executing instructions and enters sleep mode. See *WFI* on page 3-185 for more information.

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.

See *WFE* on page 3-184 for more information.

If the event register is 1, this indicates that the processor must not enter sleep mode on execution of a WFE instruction. Typically, this is because an external event signal, if implemented, is asserted, or a processor in the system has executed an SEV instruction, see *SEV* on page 3-182. Software cannot access this register directly.

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 returns to Thread mode and immediately enters sleep mode. Use this mechanism in applications that only require the processor to run when an exception occurs.

2.5.2   Wakeup from sleep mode

The conditions for the processor to wakeup depend on the mechanism that cause it to enter sleep mode.

Wakeup from WFI or sleep-on-exit

Normally, the processor wakes up only when it detects an exception with sufficient priority to cause exception entry. Some embedded systems might have to execute system restore tasks after the processor wakes up, and before it executes an interrupt handler. To achieve this set the PRIMASK bit to 1 and the FAULTMASK bit to 0. If an interrupt arrives that is enabled and has a higher priority than the current exception priority, the processor wakes up but does not execute the interrupt handler until the processor sets PRIMASK to zero. For more information about PRIMASK and FAULTMASK see *Exception mask registers* on page 2-7.

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 if implemented, see *The optional external event input*.
*   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. For more information about the SCR see *System Control Register* on page 4-20.

2.5.3   The optional Wakeup Interrupt Controller

Your device might include a *Wakeup Interrupt Controller* (WIC), an optional peripheral that can detect an interrupt and wake the processor from deep sleep mode. The WIC is enabled only when the DEEPSLEEP bit in the SCR is set to 1, see *System Control Register* on page 4-20.

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-M7 processor. This has the side effect of stopping the SysTick timer. When the WIC receives an interrupt, it takes a number of 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.

2.5.4   The optional external event input

Your device might include 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 one to indicate that the processor must not enter sleep mode on a later WFE instruction. See *Wait for event* on page 2-30 for more information.

2.5.5 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
```

Chapter 3
The Cortex-M7 Instruction Set

This chapter describes the Cortex-M7 instruction set. The following sections give general information:

- *Instruction set summary* on page 3-2.
- *CMSIS functions* on page 3-12.
- *About the instruction descriptions* on page 3-14.

Each of the following sections describes a functional group of Cortex-M7 instructions. Together they describe all the instructions supported by the Cortex-M7 processor:

- *Memory access instructions* on page 3-24.
- *General data processing instructions* on page 3-42.
- *Multiply and divide instructions* on page 3-82.
- *Saturating instructions* on page 3-104.
- *Packing and unpacking instructions* on page 3-116.
- *Bit field instructions* on page 3-123.
- *Branch and control instructions* on page 3-127.
- *Floating-point instructions* on page 3-135.
- *Miscellaneous instructions* on page 3-172.

3.1   Instruction set summary

The processor implements a version of the Thumb instruction set. Table 3-1 lists the supported instructions.

──── **Note** ────

In Table 3-1:

- 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 3-1 Cortex-M7 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry	N,Z,C,V	page 3-44
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	page 3-44
ADD, ADDW	{Rd,} Rn, #imm12	Add	-	page 3-44
ADR	Rd, label	Address to Register	-	page 3-25
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	page 3-47
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right	N,Z,C	page 3-49
B	label	Branch	-	page 3-128
BFC	Rd, #lsb, #width	Bit Field Clear	-	page 3-124
BFI	Rd, Rn, #lsb, #width	Bit Field Insert	-	page 3-124
BIC, BICS	{Rd,} Rn, Op2	Bit Clear	N,Z,C	page 3-47
BKPT	#imm8	Breakpoint	-	page 3-173
BL	label	Branch with Link	-	page 3-128
BLX	Rm	Branch indirect with Link and Exchange	-	page 3-128
BX	Rm	Branch indirect and Exchange	-	page 3-128
CBNZ	Rn, label	Compare and Branch if Non Zero	-	page 3-130
CBZ	Rn, label	Compare and Branch if Zero	-	page 3-130
CLREX	-	Clear Exclusive	-	page 3-41
CLZ	Rd, Rm	Count Leading Zeros	-	page 3-51
CMN	Rn, Op2	Compare Negative	N,Z,C,V	page 3-52
CMP	Rn, Op2	Compare	N,Z,C,V	page 3-52
CPS	Rd, Rn	Change Processor State	-	page 3-174
CPY	Rd, Rn	Copy	-	page 3-174

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
DMB	{opt}	Data Memory Barrier	-	page 3-175
DSB	{opt}	Data Synchronization Barrier	-	page 3-176
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	page 3-47
ISB	{opt}	Instruction Synchronization Barrier	-	page 3-177
IT	-	If-Then condition block	-	page 3-131
LDM	Rn{!}, reglist	Load Multiple registers	-	page 3-34
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple, decrement before	-	page 3-34
LDMIA, LDMFD	Rn{!}, reglist	Load Multiple registers, increment after	-	page 3-34
LDR, LDRT	Rt, [Rn, #offset]	Load Register with word (immediate offset, unprivileged)	-	page 3-26, page 3-31
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register with Halfword (immediate offset, unprivileged)	-	page 3-26, page 3-31
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Halfword (immediate offset, unprivileged)	-	page 3-26, page 3-31
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with byte (immediate offset, unprivileged)	-	page 3-26, page 3-31
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte (immediate offset, unprivileged)	-	page 3-26, page 3-31
LDR	Rt, [Rn, Rm {, LSL #shift}]	Load Register with word (register offset)	-	page 3-29
LDRH	Rt, [Rn, Rm {, LSL #shift}]	Load Register with Halfword (register offset)	-	page 3-29
LDRSH	Rt, [Rn, Rm {, LSL #shift}]	Load Register with Signed Halfword (register offset)	-	page 3-29
LDRB	Rt, [Rn, Rm {, LSL #shift}]	Load Register with Byte (register offset)	-	page 3-29
LDRSB	Rt, [Rn, Rm {, LSL #shift}]	Load Register with Signed Byte (register offset)	-	page 3-29
LDR	Rt, label	Load Register with word (literal)	-	page 3-32
LDRH	Rt, label	Load Register with Halfword (literal)	-	page 3-32
LDRB	Rt, label	Load Register with Byte (literal)	-	page 3-32
LDRD	Rt, Rt2, [Rn, #offset]	Load Register Dual with two bytes (immediate offset)	-	page 3-26
LDRD	Rt, Rt2, label	Load Register Dual with two bytes (PC-relative)	-	page 3-32
LDREX	Rt, [Rn, #offset]	Load Register Exclusive	-	page 3-39
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte	-	page 3-39
LDREXH	Rt, [Rn]	Load Register Exclusive with Halfword	-	page 3-39

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
LDRSB	Rt, label	Load Register with Signed Byte (PC-relative)	-	page 3-32
LDRSH	Rt, label	Load Register with Signed Halfword (PC-relative)	-	page 3-32
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical Shift Left	N,Z,C	page 3-49
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical Shift Right	N,Z,C	page 3-49
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate, 32-bit result	N,Z	page 3-83
MLS	Rd, Rn, Rm, Ra	Multiply and Subtract, 32-bit result	-	page 3-83
MOV, MOVS	Rd, Op2	Move	N,Z,C	page 3-53
MOV, MOVS	Rd, Rm	Move (register)	N,Z	page 3-53
MOVT	Rd, #imm16	Move Top	-	page 3-55
MOVW	Rd, #imm16	Move 16-bit constant	N,Z,C	page 3-53
MRS	Rd, spec_reg	Move from Special Register to general register	-	page 3-178
MSR	spec_reg, Rn	Move from general register to Special Register	-	page 3-179
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z	page 3-83
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C	page 3-53
NEG	{Rd,} Rm	Negate	-	page 3-180
NOP	-	No Operation	-	page 3-181
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	page 3-47
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	page 3-47
PKHTB, PKHBT	{Rd,} Rn, Rm, {, Op2}	Pack Halfword	-	page 3-117
PLD	[Rn {, #offset}]	Preload Data	-	page 3-36
POP	reglist	Pop registers from stack	-	page 3-37
PUSH	reglist	Push registers onto stack	-	page 3-37
QADD	{Rd,} Rn, Rm	Saturating double and Add	Q	page 3-107
QADD16	{Rd,} Rn, Rm	Saturating Add 16	-	page 3-107
QADD8	{Rd,} Rn, Rm	Saturating Add 8	-	page 3-107
QASX	{Rd,} Rn, Rm	Saturating Add and Subtract with Exchange	-	page 3-109
QDADD	{Rd,} Rn, Rm	Saturating Double and Add	Q	page 3-111
QDSUB	{Rd,} Rn, Rm	Saturating Double and Subtract	Q	page 3-111
QSAX	{Rd,} Rn, Rm	Saturating Subtract and Add with Exchange	-	page 3-109

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
QSUB	{Rd,} Rn, Rm	Saturating Subtract	Q	page 3-107
QSUB16	{Rd,} Rn, Rm	Saturating Subtract 16	-	page 3-107
QSUB8	{Rd,} Rn, Rm	Saturating Subtract 8	-	page 3-107
RBIT	Rd, Rn	Reverse Bits	-	page 3-56
REV	Rd, Rn	Reverse byte order in a word	-	page 3-56
REV16	Rd, Rn	Reverse byte order in each halfword	-	page 3-56
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-	page 3-56
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right	N,Z,C	page 3-49
RRX, RRXS	Rd, Rm	Rotate Right with Extend	N,Z,C	page 3-49
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract	N,Z,C,V	page 3-44
SADD16	{Rd,} Rn, Rm	Signed Add 16	GE	page 3-57
SADD8	{Rd,} Rn, Rm	Signed Add 8	GE	page 3-57
SASX	{Rd,} Rn, Rm	Signed Add and Subtract with Exchange	GE	page 3-65
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry	N,Z,C,V	page 3-44
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract	-	page 3-125
SDIV	{Rd,} Rn, Rm	Signed Divide	-	page 3-103
SEL	{Rd,} Rn, Rm	Select bytes	GE	page 3-77
SEV	-	Send Event	-	page 3-182
SHADD16	{Rd,} Rn, Rm	Signed Halving Add 16	-	page 3-59
SHADD8	{Rd,} Rn, Rm	Signed Halving Add 8	-	page 3-59
SHASX	{Rd,} Rn, Rm	Signed Halving Add and Subtract with Exchange	-	page 3-60
SHSAX	{Rd,} Rn, Rm	Signed Halving Subtract and Add with Exchange	-	page 3-60
SHSUB16	{Rd,} Rn, Rm	Signed Halving Subtract 16	-	page 3-62
SHSUB8	{Rd,} Rn, Rm	Signed Halving Subtract 8	-	page 3-62
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate halfwords	Q	page 3-87
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed Multiply Accumulate Dual	Q	page 3-89
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate Long $(32 \times 32 + 64)$, 64-bit result	-	page 3-102
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long, halfwords	-	page 3-91

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed Multiply Accumulate Long Dual	-	page 3-91
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed Multiply Accumulate, word by halfword	Q	page 3-87
SMLSD, SMLSDX	Rd, Rn, Rm, Ra	Signed Multiply Subtract Dual	Q	page 3-93
SMLSLD, SMLSLDX	RdLo, RdHi, Rn, Rm	Signed Multiply Subtract Long Dual	-	page 3-93
SMMLA, SMMLAR	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Accumulate	-	page 3-95
SMMLS, SMMLSR	Rd, Rn, Rm, Ra	Signed Most significant word Multiply Subtract	-	page 3-95
SMMUL, SMMULR	{Rd,} Rn, Rm	Signed Most significant word Multiply	-	page 3-97
SMUAD, SMUADX	{Rd,} Rn, Rm	Signed Dual Multiply Add	Q	page 3-98
SMULBB, SMULBT, SMULTB, SMULTT	{Rd,} Rn, Rm	Signed Multiply (halfwords)	-	page 3-100
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply Long (32 × 32), 64-bit result	-	page 3-102
SMULWB, SMULWT	{Rd,} Rn, Rm	Signed Multiply word by halfword	-	page 3-100
SMUSD, SMUSDX	{Rd,} Rn, Rm	Signed Dual Multiply Subtract	-	page 3-98
SSAT	Rd, #n, Rm {,shift #s}	Signed Saturate	Q	page 3-105
SSAT16	Rd, #n, Rm	Signed Saturate 16	Q	page 3-106
SSAX	{Rd,} Rn, Rm	Signed Subtract and Add with Exchange	GE	page 3-65
SSUB16	{Rd,} Rn, Rm	Signed Subtract 16	GE	page 3-63
SSUB8	{Rd,} Rn, Rm	Signed Subtract 8	GE	page 3-63
STM	Rn{!}, reglist	Store Multiple registers	-	page 3-34
STMDB, STMEA	Rn{!}, reglist	Store Multiple registers, decrement before	-	page 3-34
STMIA, STMFD	Rn{!}, reglist	Store Multiple registers, increment after	-	page 3-34
STR, STRT	Rt, [Rn, #offset]	Store Register word (immediate offset, unprivileged)	-	page 3-26, page 3-31
STRH, STRHT	Rt, [Rn, #offset]	Store Register Halfword (immediate offset, unprivileged)	-	page 3-26, page 3-31
STRB, STRBT	Rt, [Rn, #offset]	Store Register Byte (immediate offset, unprivileged)	-	page 3-26, page 3-31
STR	Rt, [Rn, Rm {, LSL #shift}]	Store Register word (register offset)	-	page 3-29
STRH	Rt, [Rn, Rm {, LSL #shift}]	Store Register Halfword (register offset)	-	page 3-29
STRB	Rt, [Rn, Rm {, LSL #shift}]	Store Register Byte (register offset)	-	page 3-29
STRD	Rt, Rt2, [Rn, #offset]	Store Register Dual two words	-	page 3-26

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
STREX	Rd, Rt, [Rn, #offset]	Store Register Exclusive	-	page 3-39
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte	-	page 3-39
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword	-	page 3-39
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	page 3-44
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	-	page 3-44
SVC	#imm	Supervisor Call	-	page 3-183
SXTAB	{Rd,} Rn, Rm {,ROR #n}	Sign extend 8 bits to 32 and Add	-	page 3-121
SXTAB16	{Rd,} Rn, Rm {,ROR #n}	Sign extend two 8-bit values to 16 and Add	-	page 3-121
SXTAH	{Rd,} Rn, Rm {,ROR #n}	Sign extend 16 bits to 32 and Add	-	page 3-121
SXTB	Rd, Rm {,ROR #n}	Sign extend 8 bits to 32	-	page 3-126
SXTB16	{Rd,} Rm {,ROR #n}	Sign extend 8 bits to 16	-	page 3-119
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a Halfword to 32	-	page 3-126
TBB	[Rn, Rm]	Table Branch Byte	-	page 3-133
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword	-	page 3-133
TEQ	Rn, Op2	Test Equivalence	N,Z,C	page 3-67
TST	Rn, Op2	Test	N,Z,C	page 3-67
UADD16	{Rd,} Rn, Rm	Unsigned Add 16	GE	page 3-68
UADD8	{Rd,} Rn, Rm	Unsigned Add 8	GE	page 3-68
UASX	{Rd,} Rn, Rm	Unsigned Add and Subtract with Exchange	GE	page 3-70
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract	-	page 3-125
UDIV	{Rd,} Rn, Rm	Unsigned Divide	-	page 3-103
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE	page 3-70
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16	-	page 3-73
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8	-	page 3-73
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange	-	page 3-74
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange	-	page 3-74
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16	-	page 3-76
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8	-	page 3-76
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long ($32 \times 32 + 32 + 32$), 64-bit result	-	page 3-85

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate Long (32 × 32 + 64), 64-bit result	-	page 3-102
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Long (32 × 32), 64-bit result	-	page 3-102
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	-	page 3-114
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	-	page 3-114
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	-	page 3-112
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	-	page 3-112
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	-	page 3-114
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	-	page 3-114
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	-	page 3-78
USADA8	Rd, Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	-	page 3-79
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q	page 3-105
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q	page 3-106
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange	GE	page 3-70
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE	page 3-80
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE	page 3-80
UXTAB	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend 8 bits to 32 and Add	-	page 3-121
UXTAB16	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend two 8-bit values to 16 and Add	-	page 3-121
UXTAH	{Rd,} Rn, Rm {,ROR #n}	Rotate, unsigned extend and Add Halfword	-	page 3-121
UXTB	Rd, Rm {,ROR #n}	Unsigned zero-extend a Byte	-	page 3-126
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned zero-extend Byte 16	-	page 3-126
UXTH	Rd, Rm {,ROR #n}	Unsigned zero-extend a Halfword	-	page 3-126
VABS	.F<32|64> <Sd|Dd>, <Sm|Dm>	Floating-point Absolute	-	page 3-137
VADD	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Add	-	page 3-138
VCMP	.F<32|64> <Sd|Dd>, <<Sm| #0.0>	Compare two floating-point registers, or one floating-point register and zero	N,Z,C,V	page 3-139
VCMPE	.F<32|64> <Sd|Dd>, <<Sm| #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	N,Z,C,V	page 3-139

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
VCVTA	.Tm.F<32|64> <Sd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding to nearest with Ties Away	-	page 3-169
VCVTN	.Tm.F<32|64> <Sd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding to nearest with Ties to even	-	page 3-169
VCVTP	.Tm.F<32|64> <Sd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding towards Plus infinity	-	page 3-169
VCVTM	.Tm.F<32|64> <Sd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding towards Minus infinity	-	page 3-169
VCVT	.F<32|64>.Tm <Sd>, <Sm|Dm>	Convert from floating-point to integer	-	page 3-140
VCVTR	.Tm.F<32|64> <Sd>, <Sm|Dm>	Convert between floating-point and integer with rounding.	-	page 3-140
VCVT	.Td.F<32|64> <Sd|Dd>, <Sd|Dd>, #fbits	Convert from floating-point to fixed point	-	page 3-141
VCVT	<B|T>.F<32|64>.F16 <Sd|Dd>, Sm	Convert half-precision value to single-precision or double-precision	-	page 3-142
VCVT	<B|T>.F16.F<32|64> Sd, <Sm|Dm>	Convert single-precision or double precision register to half-precision	-	page 3-142
VDIV	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Divide	-	page 3-143
VFMA	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Multiply Accumulate	-	page 3-144
VFMS	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Multiply Subtract	-	page 3-144
VFNMA	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Negate Multiply Accumulate	-	page 3-145
VFNMS	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Negate Multiply Subtract	-	page 3-145
VLDM	{mode}{.size} Rn{!}, list	Floating-point Load Multiple extension registers	-	page 3-146
VLDR	.F<32|64> <Sd|Dd>, [<Rn> {, #offset}]	Floating-point Load an extension register from memory (immediate)	-	page 3-147
VLDR.F<32|64>	<Sd|Dd>, <label>	Load an extension register from memory	-	page 3-147
VLDR.F<32|64>	<Sd|Dd>, [PC,#-0]	Load an extension register from memory	-	page 3-147
VMAXNM	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Maximum of two floating-point numbers with IEEE754-2008 NaN handling	-	page 3-168
VMINNM	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Minimum of two floating-point numbers with IEEE754-2008 NaN handling	-	page 3-168
VMLA	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Floating-point Multiply Accumulate	-	page 3-148
VMLS	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Floating-point Multiply Subtract	-	page 3-148

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
VMRS	Rt, FPSCR	Move to ARM core register from floating-point Special Register	N,Z,C,V	page 3-156
VMSR	FPSCR, Rt	Move to floating-point Special Register from ARM core register	-	page 3-157
VMOV	<Sn|Rt>, <Rt|Sn>	Copy ARM core register to single-precision	-	page 3-152
VMOV	<Sm|Rt>, <Sm1|Rt2>, <Rt|Sm>, <Rt2|Sm1>	Copy two ARM core registers to two single-precision	-	page 3-153
VMOV	{.size} Dd[x], Rt	Copy ARM core register to scalar	-	page 3-155
VMOV	{.dt} Rt, Dn[x]	Copy scalar to ARM core register	-	page 3-151
VMOV	.F<32|64> <Sd|Dd>, #immm	Floating-point Move immediate	-	page 3-149
VMOV	.F<32|64> <Sd|Dd>, <Sd|Dd>, <Sm|Dm>	Copies the contents of one register to another	-	page 3-150
VMOV	<Dm|Rt>, <Rt|Rt2>, <Rt2|Dm>	Floating-point Move transfers two words between two ARM core registers and a doubleword register	-	page 3-154
VMUL	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Multiply	-	page 3-158
VNEG	.F<32|64> <Sd|Dd>, <Sm|Dm>	Floating-point Negate	-	page 3-159
VNMLA	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Floating-point Multiply Accumulate and Negate	-	page 3-160
VNMLS	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Floating-point Multiply, Subtract and Negate	-	page 3-160
VNMUL	.F<32|64> {<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Multiply and Negate	-	page 3-160
VPOP	{.size} list	Load multiple consecutive floating-point registers from the stack	-	page 3-161
VPUSH	{.size} list	Store multiple consecutive floating-point registers to the stack	-	page 3-162
VRINTA	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties Away	-	page 3-171
VRINTN	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with directed rounding to Nearest with Ties to even	-	page 3-171
VRINTP	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with directed rounding to Plus infinity	-	page 3-171
VRINTM	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with directed rounding to Minus infinity	-	page 3-171

Table 3-1 Cortex-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
VRINTX	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with rounding specified in FPSCR	-	page 3-170
VRINTZ	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with rounding towards Zero	-	page 3-171
VRINTR	.F<32|64> <Sd|Dd>, <Sm|Dm>	Float to integer in floating-point format conversion with rounding towards value specified in FPSCR	-	page 3-170
VSEL	.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>	Select register, alternative to a pair of conditional VMOV	-	page 3-167
VSQRT	.F<32|64> <Sd|Dd>, <Sm|Dm>	Calculates floating-point Square Root	-	page 3-163
VSTM	{mode}{.size} Rn{!}, list	Floating-point Store Multiple	-	page 3-164
VSTR	.F<32|64> <Sd|Dd>, [Rn{, #offset}]	Floating-point Store Register stores an extension register to memory	-	page 3-165
VSUB.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Subtract	-	page 3-166
WFE	-	Wait For Event	-	page 3-184
WFI	-	Wait For Interrupt	-	page 3-185

3.1.1 Binary compatibility with other Cortex processors

The processor implements the ARMv7-M instruction set and features provided by the ARMv7-M architecture profile, and is binary compatible with the instruction sets and features implemented in other Cortex-M profile processors. You cannot move software from the Cortex-M7 processor to:

- The Cortex-M3 processor if it contains floating-point operations or DSP extensions.

- The Cortex-M4 processor if it contains double-precision floating-point operations.

- The Cortex-M0 or Cortex-M0+ processors because these are implementations of the ARMv6-M Architecture.

Code designed for other Cortex-M processors is compatible with Cortex-M7 as long as it does not rely on bit-banding.

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.

3.2 CMSIS functions

ISO/IEC C code cannot directly access some Cortex-M7 processor instructions. This section describes intrinsic functions that can generate these instructions, provided by the CMSIS and that might be provided by a C compiler. If a C compiler does not support an appropriate intrinsic function, you might have to use inline assembler to access some instructions.

The CMSIS provides the following intrinsic functions to generate instructions that ISO/IEC C code cannot directly access:

Table 3-2 CMSIS functions to generate some Cortex-M7 processor instructions

Instruction	CMSIS function
CPSIE I	void __enable_irq(void)
CPSID I	void __disable_irq(void)
CPSIE F	void __enable_fault_irq(void)
CPSID F	void __disable_fault_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
RBIT	uint32_t __RBIT(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

The CMSIS also provides a number of functions for accessing the special registers using MRS and MSR instructions:

Table 3-3 CMSIS functions to access the special registers

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)

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

Special register	Access	CMSIS function
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)`

3.3 About the instruction descriptions

The following sections give more information about using the instructions:

- *Operands* on page 3-15.
- *Restrictions when using PC or SP* on page 3-15.
- *Flexible second operand* on page 3-15.
- *Shift Operations* on page 3-16.
- *Address alignment* on page 3-19.
- *PC-relative expressions* on page 3-19.
- *Conditional execution* on page 3-20.
- *Instruction width selection* on page 3-23.

3.3.1 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. See *Flexible second operand*.

3.3.2 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.

——— **Note** ———

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-M7 processor only supports Thumb instructions.

3.3.3 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*.
* *Register with optional shift* on page 3-16.

Constant

You specify an Operand2 constant in the form:

`#constant`

where `constant` can be:

* Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.

* Any constant of the form `0x00XY00XY`.

* Any constant of the form `0xXY00XY00`.

* Any constant of the form `0xXYXYXYXY`.

——— **Note** ———

In these constants, X and Y are hexadecimal digits.

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

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

Instruction substitution

Your assembler might be able to produce an equivalent instruction in cases where you specify a constant that is not permitted. For example, an assembler might assemble the instruction CMP *Rd*, #0xFFFFFFFE as the equivalent instruction CMN *Rd*, #0x2.

Register with optional shift

You specify an Operand2 register in the form:

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   Rotate right one bit, with extend.

-              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. For information on the shift operations and how they affect the carry flag, see *Shift Operations*.

3.3.4   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, see *Flexible second operand* on page 3-15. 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 *Flexible second operand* on page 3-15. 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.

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. See Figure 3-1 on page 3-17.

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*.

> **Note**
>
> - 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*.
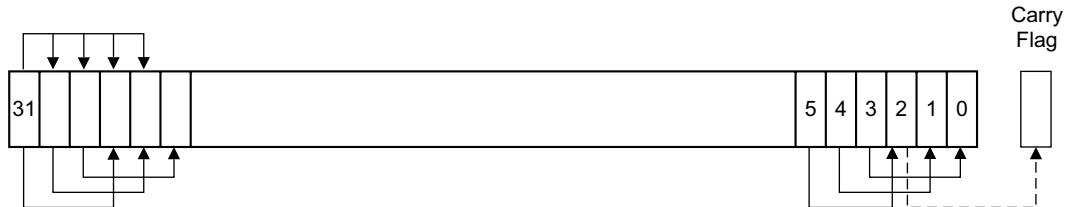
Figure 3-1 ASR #3

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. See Figure 3-2.

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*.

> **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.
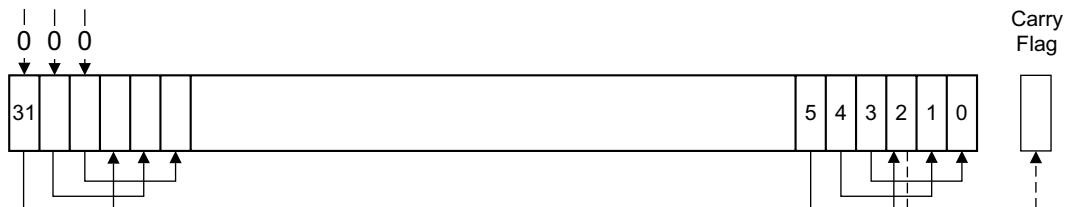
Figure 3-2 LSR #3

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. See Figure 3-3 on page 3-18.

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.
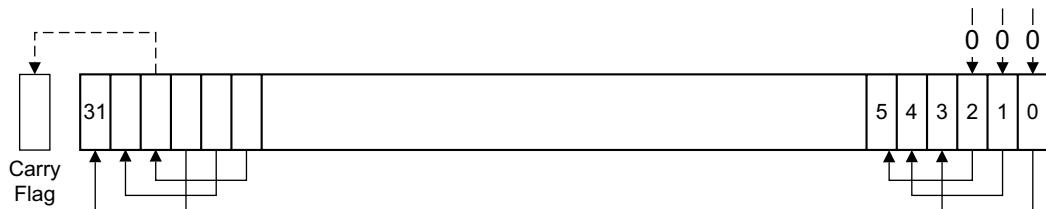
Figure 3-3 LSL #3

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. See Figure 3-4.

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.
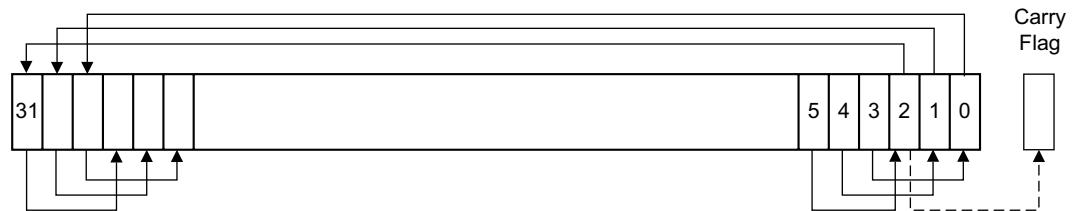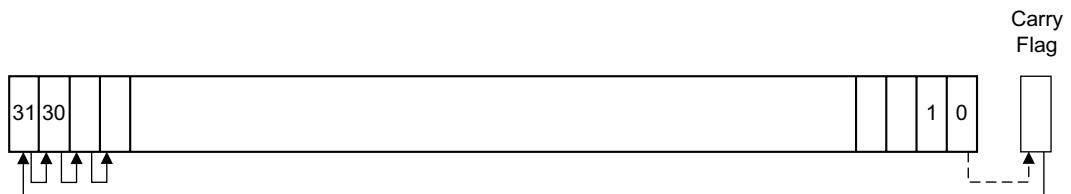
Figure 3-4 ROR #3

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. See Figure 3-5 on page 3-19.

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 3-5 RRX

3.3.5 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-M7 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. For more information about UsageFaults see *Fault handling* on page 2-27.

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, see *Configuration and Control Register* on page 4-20.

3.3.6 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.

——— **Note** ———

* For B, BL, CBNZ, and CBZ instructions, the value of the PC is the address of the current instruction plus 4 bytes.

* For all other instructions that use labels, the value of the PC is the address of the current instruction plus 4 bytes, with bit[1] of the result cleared to 0 to make it word-aligned.

* Your assembler might permit other syntaxes for PC-relative expressions, such as a label plus or minus a number, or an expression of the form [PC, #number].

3.3.7 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, see *Application Program Status Register* on page 2-5. 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. See Table 3-4 on page 3-21 for a list of the suffixes to add to instructions to make them conditional 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. See *IT* on page 3-131 for more information and restrictions when using the IT instruction. Depending on the vendor, the assembler might automatically insert an IT instruction if you have conditional instructions outside the IT block.

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

This section describes:
- *The condition flags* on page 3-21.
- *Condition code suffixes* on page 3-21.

The condition flags

The APSR contains the following 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 the APSR see *Program Status Register* on page 2-4.

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:

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

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

——— **Note** ———

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

————————

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. Table 3-4 shows the condition codes to use.

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

Table 3-4 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 3-4 Condition code suffixes

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

Table 3-4 Condition code suffixes (continued)

Suffix	Flags	Meaning
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.
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.

Example 3-1 shows the use of a conditional instruction to find the absolute value of a number. R0 = abs(R1).

Example 3-1 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.
```

Example 3-2 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.

Example 3-2 Compare and update value

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

3.3.8 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. Example 3-3 shows instructions with the instruction width suffix.

Example 3-3 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.
```

3.4 Memory access instructions

Table 3-5 shows the memory access instructions:

Table 3-5 Memory access instructions

Mnemonic	Brief description	See
ADR	Generate PC-relative address	*ADR* on page 3-25
CLREX	Clear Exclusive	*CLREX* on page 3-41
LDM{mode}	Load Multiple registers	*LDM and STM* on page 3-34
LDR{type}	Load Register using immediate offset	*LDR and STR, immediate offset* on page 3-26
LDR{type}	Load Register using register offset	*LDR and STR, register offset* on page 3-29
LDR{type}T	Load Register with unprivileged access	*LDR and STR, unprivileged* on page 3-31
LDR	Load Register using PC-relative address	*LDR, PC-relative* on page 3-32
LDRD	Load Register Dual	*LDR and STR, immediate offset* on page 3-26
LDREX{type}	Load Register Exclusive	*LDREX and STREX* on page 3-39
PLD	Preload Data.	*PLD* on page 3-36
POP	Pop registers from stack	*PUSH and POP* on page 3-37
PUSH	Push registers onto stack	*PUSH and POP* on page 3-37
STM{mode}	Store Multiple registers	*LDM and STM* on page 3-34
STR{type}	Store Register using immediate offset	*LDR and STR, immediate offset* on page 3-26
STR{type}	Store Register using register offset	*LDR and STR, register offset* on page 3-29
STR{type}T	Store Register with unprivileged access	*LDR and STR, unprivileged* on page 3-31
STREX{type}	Store Register Exclusive	*LDREX and STREX* on page 3-39

3.4.1 ADR

Generate PC-relative address.

Syntax

`ADR{`*`cond`*`}` *`Rd`*`,` *`label`*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

Rd   Is the destination register.

label   Is a PC-relative expression. See *PC-relative expressions* on page 3-19.

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. See *Instruction width selection* on page 3-23.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

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

3.4.2   LDR and STR, immediate offset

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

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]       ; immediate offset

op{type}{cond} Rt, [Rn, #offset]!         ; pre-indexed

op{type}{cond} Rt, [Rn], #offset          ; post-indexed

opD{cond} Rt, Rt2, [Rn {, #offset}]       ; immediate offset, two words

opD{cond} Rt, Rt2, [Rn, #offset]!         ; pre-indexed, two words

opD{cond} Rt, Rt2, [Rn], #offset          ; post-indexed, two words
```

Where:

op   Is one of:
LDR	Load Register.
STR	Store Register.

type   Is one of:
B	Unsigned byte, zero extend to 32 bits on loads.
SB	Signed byte, sign extend to 32 bits (LDR only).
H	Unsigned halfword, zero extend to 32 bits on loads.
SH	Signed halfword, sign extend to 32 bits (LDR only).
-	Omit, for word.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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. See *Address alignment* on page 3-19.

Table 3-6 shows the ranges of offset for immediate, pre-indexed and post-indexed forms.

Table 3-6 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.

Examples

```
LDR     R8, [R10]            ; Loads R8 from the address in R10.
LDRNE   R2, [R5, #960]!      ; Loads (conditionally) R2 from a word
                             ; 960 bytes above the address in R5, and
                             ; increments R5 by 960.
STR     R2, [R9,#const-struc] ; const-struc is an expression evaluating
                             ; to a constant in the range 0-4095.
STRH    R3, [R4], #4         ; Store R3 as halfword data into address in
                             ; R4, then increment R4 by 4.
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.
```

3.4.3   LDR and STR, register offset

Load and Store with register offset.

Syntax

op{*type*}{*cond*} *Rt*, [*Rn*, *Rm* {, LSL #*n*}]

Where:

op	Is one of:	
	LDR	Load Register.
	STR	Store Register.

type	Is one of:	
	B	Unsigned byte, zero extend to 32 bits on loads.
	SB	Signed byte, sign extend to 32 bits (LDR only).
	H	Unsigned halfword, zero extend to 32 bits on loads.
	SH	Signed halfword, sign extend to 32 bits (LDR only).
	-	omit, for word.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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. See *Address alignment* on page 3-19.

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.

Examples

```
STR    R0, [R5, R1]         ; Store value of R0 into an address equal to
                            ; sum of R5 and R1.
LDRSB  R0, [R5, R1, LSL #1] ; Read byte value from an address equal to
                            ; sum of R5 and two times R1, sign extended it
                            ; to a word value and put it in R0.
STR    R0, [R1, R2, LSL #2] ; Stores R0 to an address equal to sum of R1
                            ; and four times R2.
```

3.4.4   LDR and STR, unprivileged

Load and Store with unprivileged access.

Syntax

op{*type*}T{*cond*} *Rt*, [*Rn* {, #*offset*}]

Where:

op	Is one of:

               LDR   Load Register.

               STR   Store Register.

type   Is one of:

               B   Unsigned byte, zero extend to 32 bits on loads.

               SB   Signed byte, sign extend to 32 bits (LDR only).

               H   Unsigned halfword, zero extend to 32 bits on loads.

               SH   Signed halfword, sign extend to 32 bits (LDR only).

               -             Omit, for word.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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, see *LDR and STR, immediate offset* on page 3-26. 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.

Examples

```
    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.
```

3.4.5 LDR, PC-relative

Load register from memory.

Syntax

LDR{*type*}{*cond*} *Rt*, *label*

LDRD{*cond*} *Rt*, *Rt2*, *label*   ; Load two words

Where:

type	Is one of:

	B	Unsigned byte, zero extend to 32 bits.
	SB	Signed byte, sign extend to 32 bits.
	H	Unsigned halfword, zero extend to 32 bits.
	SH	Signed halfword, sign extend to 32 bits.
	-	Omit, for word.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
Rt	Is the register to load or store.
Rt2	Is the second register to load or store.
label	Is a PC-relative expression. See *PC-relative expressions* on page 3-19.

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. See *Address alignment* on page 3-19.

label must be within a limited range of the current instruction. Table 3-7 shows the possible offsets between *label* and the PC.

Table 3-7 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. See *Instruction width selection* on page 3-23.

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.

Examples

```
LDR     R0, LookUpTable   ; Load R0 with a word of data from an address
                          ; labelled as LookUpTable.
LDRSB   R7, localdata     ; Load a byte value from an address labelled
                          ; as localdata, sign extend it to a word
                          ; value, and put it in R7.
```

3.4.6   LDM and STM

Load and Store Multiple registers.

Syntax

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. See *Conditional execution* on page 3-20.

Rn   Is the register on which the memory addresses are based.

!   Is an optional writeback 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, see *Examples* on page 3-35.

`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 writeback 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 writeback 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. See *PUSH and POP* on page 3-37 for details.

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 writeback 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.

Examples

```
LDM     R8,{R0,R2,R9}      ; LDMIA is a synonym for LDM.
STMDB   R1!,{R3-R6,R11,R12}
```

Incorrect examples

```
STM     R5!,{R5,R4,R9} ; Value stored for R5 is unpredictable.
LDM     R2, {}         ; There must be at least one register in the list.
```

3.4.7   PLD

Preload Data.

Syntax

```
PLD{cond} [Rn {, #imm}]              ; Immediate

PLD{cond} [Rn, Rm {, LSL #shift}]  ; Register

PLD{cond} label                      ; Literal
```

where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.4.8   PUSH and POP

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

Syntax

PUSH{*cond*} *reglist*

POP{*cond*} *reglist*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

See *LDM and STM* on page 3-34 for more information.

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.

Examples

```
PUSH {R0,R4-R7} ; Push R0,R4,R5,R6,R7 onto the stack
PUSH {R2,LR}    ; Push R2 and the link-register onto the stack
POP  {R0,R6,PC} ; Pop r0,r6 and PC from the stack, then branch to the new PC.
```

3.4.9 LDREX and STREX

Load and Store Register Exclusive.

Syntax

LDREX{*cond*} *Rt*, [*Rn* {, #*offset*}]

STREX{*cond*} *Rd*, *Rt*, [*Rn* {, #*offset*}]

LDREXB{*cond*} *Rt*, [*Rn*]

STREXB{*cond*} *Rd*, *Rt*, [*Rn*]

LDREXH{*cond*} *Rt*, [*Rn*]

STREXH{*cond*} *Rd*, *Rt*, [*Rn*]

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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, see *Synchronization primitives* on page 2-16.

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.

Examples

```
    MOV    R1, #0x1           ; Initialize the 'lock taken' value
try
    LDREX  R0, [LockAddr]     ; Load the lock value
    CMP    R0, #0             ; Is the lock free?
    ITT    EQ                 ; IT instruction for STREXEQ and CMPEQ
    STREXEQ R0, R1, [LockAddr]  ; Try and claim the lock
    CMPEQ  R0, #0             ; Did this succeed?
    BNE    try                ; No – try again
    ....                      ; Yes – we have the lock.
```

3.4.10   CLREX

Clear Exclusive.

Syntax

CLREX{*cond*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

Operation

Use CLREX to make the next STREX, STREXB, or STREXH instruction write 1 to its destination register and fail to perform the store. It is useful in exception handler code 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.

See *Synchronization primitives* on page 2-16 for more information.

Condition flags

This instruction does not change the flags.

Examples

CLREX

3.5 General data processing instructions

Table 3-8 shows the data processing instructions:

Table 3-8 Data processing instructions

Mnemonic	Brief description	See
ADC	Add with Carry	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
ADD	Add	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
ADDW	Add	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
AND	Logical AND	*AND, ORR, EOR, BIC, and ORN* on page 3-47
ASR	Arithmetic Shift Right	*ASR, LSL, LSR, ROR, and RRX* on page 3-49
BIC	Bit Clear	*AND, ORR, EOR, BIC, and ORN* on page 3-47
CLZ	Count leading zeros	*CLZ* on page 3-51
CMN	Compare Negative	*CMP and CMN* on page 3-52
CMP	Compare	*CMP and CMN* on page 3-52
EOR	Exclusive OR	*AND, ORR, EOR, BIC, and ORN* on page 3-47
LSL	Logical Shift Left	*ASR, LSL, LSR, ROR, and RRX* on page 3-49
LSR	Logical Shift Right	*ASR, LSL, LSR, ROR, and RRX* on page 3-49
MOV	Move	*MOV and MVN* on page 3-53
MOVT	Move Top	*MOVT* on page 3-55
MOVW	Move 16-bit constant	*MOV and MVN* on page 3-53
MVN	Move NOT	*MOV and MVN* on page 3-53
ORN	Logical OR NOT	*AND, ORR, EOR, BIC, and ORN* on page 3-47
ORR	Logical OR	*AND, ORR, EOR, BIC, and ORN* on page 3-47
RBIT	Reverse Bits	*REV, REV16, REVSH, and RBIT* on page 3-56
REV	Reverse byte order in a word	*REV, REV16, REVSH, and RBIT* on page 3-56
REV16	Reverse byte order in each halfword	*REV, REV16, REVSH, and RBIT* on page 3-56
REVSH	Reverse byte order in bottom halfword and sign extend	*REV, REV16, REVSH, and RBIT* on page 3-56
ROR	Rotate Right	*ASR, LSL, LSR, ROR, and RRX* on page 3-49
RRX	Rotate Right with Extend	*ASR, LSL, LSR, ROR, and RRX* on page 3-49
RSB	Reverse Subtract	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
SADD16	Signed Add 16	*SADD16 and SADD8* on page 3-57
SADD8	Signed Add 8	*SADD16 and SADD8* on page 3-57
SASX	Signed Add and Subtract with Exchange	*SASX and SSAX* on page 3-65
SSAX	Signed Subtract and Add with Exchange	*SASX and SSAX* on page 3-65
SBC	Subtract with Carry	*ADD, ADC, SUB, SBC, and RSB* on page 3-44

Table 3-8 Data processing instructions (continued)

Mnemonic	Brief description	See
SHADD16	Signed Halving Add 16	*SHADD16 and SHADD8* on page 3-59
SHADD8	Signed Halving Add 8	*SHADD16 and SHADD8* on page 3-59
SHASX	Signed Halving Add and Subtract with Exchange	*SHASX and SHSAX* on page 3-60
SHSAX	Signed Halving Subtract and Add with Exchange	*SHASX and SHSAX* on page 3-60
SHSUB16	Signed Halving Subtract 16	*SHSUB16 and SHSUB8* on page 3-62
SHSUB8	Signed Halving Subtract 8	*SHSUB16 and SHSUB8* on page 3-62
SSUB16	Signed Subtract 16	*SSUB16 and SSUB8* on page 3-63
SSUB8	Signed Subtract 8	*SSUB16 and SSUB8* on page 3-63
SUB	Subtract	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
SUBW	Subtract	*ADD, ADC, SUB, SBC, and RSB* on page 3-44
TEQ	Test Equivalence	*TST and TEQ* on page 3-67
TST	Test	*TST and TEQ* on page 3-67
UADD16	Unsigned Add 16	*UADD16 and UADD8* on page 3-68
UADD8	Unsigned Add 8	*UADD16 and UADD8* on page 3-68
UASX	Unsigned Add and Subtract with Exchange	*UASX and USAX* on page 3-70
USAX	Unsigned Subtract and Add with Exchange	*UASX and USAX* on page 3-70
UHADD16	Unsigned Halving Add 16	*UHADD16 and UHADD8* on page 3-73
UHADD8	Unsigned Halving Add 8	*UHADD16 and UHADD8* on page 3-73
UHASX	Unsigned Halving Add and Subtract with Exchange	*UHASX and UHSAX* on page 3-74
UHSAX	Unsigned Halving Subtract and Add with Exchange	*UHASX and UHSAX* on page 3-74
UHSUB16	Unsigned Halving Subtract 16	*UHSUB16 and UHSUB8* on page 3-76
UHSUB8	Unsigned Halving Subtract 8	*UHSUB16 and UHSUB8* on page 3-76
USAD8	Unsigned Sum of Absolute Differences	*USAD8* on page 3-78
USADA8	Unsigned Sum of Absolute Differences and Accumulate	*USADA8* on page 3-79
USUB16	Unsigned Subtract 16	*USUB16 and USUB8* on page 3-80
USUB8	Unsigned Subtract 8	*USUB16 and USUB8* on page 3-80

3.5.1 ADD, ADC, SUB, SBC, and RSB

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

Syntax

```
op{S}{cond} {Rd,} Rn, Operand2          ; ADD; ADC; SBC; RSB

op{S|W}{cond} {Rd,} Rn, #imm12          ; ADD; SUB
```

Where:

op	Is one of:
	ADD Add.
	ADC Add with Carry.
	SUB Subtract.
	SBC Subtract with Carry.
	RSB Reverse Subtract.

S Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-20.

cond Is an optional condition code. See *Conditional execution* on page 3-20.

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. See *Flexible second operand* on page 3-15 for details of the options.

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, see *Multiword arithmetic examples* on page 3-45.

See also *ADR* on page 3-25.

——— **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.

Examples

```
ADD     R2, R1, R3
SUBS    R8, R6, #240        ; Sets the flags on the result.
RSB     R4, R4, #1280       ; Subtracts contents of R4 from 1280.
ADCHI   R11, R0, R3         ; Only executed if C flag set and Z.
                            ; flag clear.
```

Multiword arithmetic examples

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.

Example 3-4 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. Example 3-5 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.

Example 3-5 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.
```

3.5.2   AND, ORR, EOR, BIC, and ORN

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

Syntax

op{S}{*cond*} {*Rd*,} *Rn*, *Operand2*

Where:

op	Is one of:

	AND	Logical AND.
	ORR	Logical OR, or bit set.
	EOR	Logical Exclusive OR.
	BIC	Logical AND NOT, or bit clear.
	ORN	Logical OR NOT.

S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-20.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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. See *Flexible second operand* on page 3-15 for details of the options.

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*, see *Flexible second operand* on page 3-15.

- Do not affect the V flag.

Examples

```
AND     R9, R2, #0xFF00
ORREQ   R2, R0, R5
ANDS    R9, R8, #0x19
EORS    R7, R11, #0x18181818
BIC     R0, R1, #0xab
ORN     R7, R11, R14, ROR #4
ORNS    R7, R11, R14, ASR #32
```

3.5.3 ASR, LSL, LSR, ROR, and RRX

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

Syntax

op{S}{*cond*} *Rd*, *Rm*, *Rs*

op{S}{*cond*} *Rd*, *Rm*, *#n*

RRX{S}{*cond*} *Rd*, *Rm*

Where:

op	Is one of:	
	ASR	Arithmetic Shift Right.
	LSL	Logical Shift Left.
	LSR	Logical Shift Right.
	ROR	Rotate Right.
S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-20.	
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, see *Shift Operations* on page 3-16.

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, see *Shift Operations* on page 3-16.

Examples

```
ASR    R7, R8, #9  ; Arithmetic shift right by 9 bits.
LSLS   R1, R2, #3  ; Logical shift left by 3 bits with flag update.
LSR    R4, R5, #6  ; Logical shift right by 6 bits.
ROR    R4, R5, R6  ; Rotate right by the value in the bottom byte of R6.
RRX    R4, R5      ; Rotate right with extend.
```

3.5.4 CLZ

Count Leading Zeros.

Syntax

```
CLZ{cond} Rd, Rm
```

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

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

3.5.5 CMP and CMN

Compare and Compare Negative.

Syntax

CMP{*cond*} *Rn*, *Operand2*

CMN{*cond*} *Rn*, *Operand2*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

Rn   Is the register holding the first operand.

Operand2   Is a flexible second operand. See *Flexible second operand* on page 3-15 for
              details of the options.

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.

Examples

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

3.5.6   MOV and MVN

Move and Move NOT.

Syntax

MOV{S}{*cond*} *Rd, Operand2*

MOV{S}{*cond*} *Rd, Rm*

MOV{W}{*cond*} *Rd, #imm16*

MVN{S}{*cond*} *Rd, Operand2*

Where:

S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-20.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
Rd	Is the destination register.
Operand2	Is a flexible second operand. See *Flexible second operand* on page 3-15 for details of the options.
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:

*   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.

Also, the MOV instruction permits additional forms of *Operand2* as synonyms for shift instructions:

*   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*.

See *ASR, LSL, LSR, ROR, and RRX* on page 3-49.

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*, see *Flexible second operand* on page 3-15.

- Do not affect the V flag.

Example

```
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.
```

3.5.7   MOVT

Move Top.

Syntax

```
MOVT{cond} Rd, #imm16
```

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
MOVT   R3, #0xF123 ; Write 0xF123 to upper halfword of R3, lower halfword
                   ; and APSR are unchanged.
```

3.5.8   REV, REV16, REVSH, and RBIT

Reverse bytes and Reverse bits.

Syntax

op{*cond*} *Rd*, *Rn*

Where:

op	Is one of:

      REV   Reverse byte order in a word.

      REV16   Reverse byte order in each halfword independently.

      REVSH   Reverse byte order in the bottom halfword, and sign extend to 32 bits.

      RBIT   Reverse the bit order in a 32-bit word.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
REV    R3, R7  ; Reverse byte order of value in R7 and write it to R3.
REV16  R0, R0  ; Reverse byte order of each 16-bit halfword in R0.
REVSH  R0, R5  ; Reverse Signed Halfword.
REVHS  R3, R7  ; Reverse with Higher or Same condition.
RBIT   R7, R8  ; Reverse bit order of value in R8 and write the result to R7.
```

3.5.9 SADD16 and SADD8

Signed Add 16 and Signed Add 8.

Syntax

op{*cond*} {*Rd,*} *Rn, Rm*

Where:

op   Is one of:

        SADD16   Performs two 16-bit signed integer additions.

        SADD8   Performs four 8-bit signed integer additions.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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';
```

Examples

```
SADD16 R1, R0      ; Adds the halfwords in R0 to the corresponding halfwords of
                   ; R1 and writes to corresponding halfword of R1.
SADD8  R4, R0, R5  ; Adds bytes of R0 to the corresponding byte in R5 and writes
                   ; to the corresponding byte in R4.
```

3.5.10 SHADD16 and SHADD8

Signed Halving Add 16 and Signed Halving Add 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	SHADD16 Signed Halving Add 16.
	SHADD8 Signed Halving Add 8.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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:
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.

The SHADD8 instruction:
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.

Examples

```
SHADD16 R1, R0     ; Adds halfwords in R0 to corresponding halfword of R1 and
                   ; writes halved result to corresponding halfword in R1.
SHADD8  R4, R0, R5 ; Adds bytes of R0 to corresponding byte in R5 and
                   ; writes halved result to corresponding byte in R4.
```

3.5.11 SHASX and SHSAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of: SHASX Add and Subtract with Exchange and Halving. SHSAX Subtract and Add with Exchange and Halving.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SHASX    R7, R4, R2    ; Adds top halfword of R4 to bottom halfword of R2
                       ; and writes halved result to top halfword of R7.
                       ; Subtracts top halfword of R2 from bottom halfword of
                       ; R4 and writes halved result to bottom halfword of R7.
SHSAX    R0, R3, R5    ; Subtracts bottom halfword of R5 from top halfword
                       ; of R3 and writes halved result to top halfword of R0.
                       ; Adds top halfword of R5 to bottom halfword of R3 and
                       ; writes halved result to bottom halfword of R0.
```

3.5.12 SHSUB16 and SHSUB8

Signed Halving Subtract 16 and Signed Halving Subtract 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op   Is one of:

     SHSUB16 Signed Halving Subtract 16.

     SHSUB8   Signed Halving Subtract 8.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

Rd   Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn   Is the first operand register.

Rm   Is the second operand register.

Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register.

The SHSUB16 instruction:
1. Subtracts each halfword of the second operand from the corresponding halfwords of the first operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halved halfword results in the destination register.

The SHSUBB8 instruction:
1. Subtracts each byte of the second operand from the corresponding byte of the first operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the corresponding signed byte results in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

```
SHSUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword
                    ; of R1 and writes to corresponding halfword of R1.
SHSUB8  R4, R0, R5  ; Subtracts bytes of R0 from corresponding byte in R5,
                    ; and writes to corresponding byte in R4.
```

3.5.13   SSUB16 and SSUB8

Signed Subtract 16 and Signed Subtract 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:

SSUB16   Performs two 16-bit signed integer subtractions.

SSUB8   Performs four 8-bit signed integer subtractions.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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:
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.

The SSUB8 instruction:
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';
```

Examples

```
SSUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of R1
                   ; and writes to corresponding halfword of R1.
SSUB8  R4, R0, R5  ; Subtracts bytes of R5 from corresponding byte in
                   ; R0, and writes to corresponding byte of R4.
```

3.5.14 SASX and SSAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:

	SASX	Signed Add and Subtract with Exchange.
	SSAX	Signed Subtract and Add with Exchange.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.
Rn	Is the first operand register.
Rm	Is the second operand register.

Operation

The SASX instruction:

1. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
2. Writes the signed result of the addition to the top halfword of the destination register.
3. Subtracts the signed bottom halfword of the second operand from the top signed halfword of the first operand.
4. Writes the signed result of the subtraction to the bottom halfword of the destination register.

The SSAX instruction:

1. Subtracts the signed bottom halfword of the second operand from the top signed halfword of the first operand.
2. Writes the signed result of the addition to the bottom halfword of the destination register.
3. Adds the signed top halfword of the first operand with the signed bottom halfword of the second operand.
4. Writes the signed result of the subtraction to the top halfword of the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions set the APSR.GE bits according to the results.

For SASX:

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    sum = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0> = diff<15:0>;
    R[d]<31:16> = sum<15:0>;
    APSR.GE<1:0> = if diff >= 0 then '11' else '00';
```

```
                    APSR.GE<3:2> = if sum >= 0 then '11' else '00';
```

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';
```

Examples

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

3.5.15 TST and TEQ

Test bits and Test Equivalence.

Syntax

TST{*cond*} *Rn*, *Operand2*

TEQ{*cond*} *Rn*, *Operand2*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
Rn	Is the first operand register.
Operand2	Is a flexible second operand. See *Flexible second operand* on page 3-15 for details of the options.

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*, see *Flexible second operand* on page 3-15.

- Do not affect the V flag.

Examples

```
TST     R0, #0x3F8  ; Perform bitwise AND of R0 value to 0x3F8,
                    ; APSR is updated but result is discarded
TEQEQ   R10, R9     ; Conditionally test if value in R10 is equal to
                    ; value in R9, APSR is updated but result is discarded.
```

3.5.16   UADD16 and UADD8

Unsigned Add 16 and Unsigned Add 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	UADD16 Performs two 16-bit unsigned integer additions.
	UADD8 Performs four 8-bit unsigned integer additions.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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';
```

Examples

```
UADD16 R1, R0      ; Adds halfwords in R0 to corresponding halfword of R1,
                   ; writes to corresponding halfword of R1.
UADD8  R4, R0, R5  ; Adds bytes of R0 to corresponding byte in R5 and writes
                   ; to corresponding byte in R4.
```

3.5.17 UASX and USAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	UASX Add and Subtract with Exchange.
	USAX Subtract and Add with Exchange.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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';
```

Examples

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

3.5.18 UHADD16 and UHADD8

Unsigned Halving Add 16 and Unsigned Halving Add 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	UHADD16 Unsigned Halving Add 16.
	UHADD8 Unsigned Halving Add 8.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
UHADD16 R7, R3     ; Adds halfwords in R7 to corresponding halfword of R3
                   ; and writes halved result to corresponding halfword in R7.
UHADD8  R4, R0, R5 ; Adds bytes of R0 to corresponding byte in R5 and writes
                   ; halved result to corresponding byte in R4.
```

3.5.19   UHASX and UHSAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	UHASX Unsigned Halving Add and Subtract with Exchange.
	UHSAX Unsigned Halving Subtract and Add with Exchange.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
UHASX    R7, R4, R2    ; Adds top halfword of R4 with bottom halfword of R2
                       ; and writes halved result to top halfword of R7.
                       ; Subtracts top halfword of R2 from bottom halfword of
                       ; R7 and writes halved result to bottom halfword of R7.
UHSAX    R0, R3, R5    ; Subtracts bottom halfword of R5 from top halfword of
                       ; R3 and writes halved result to top halfword of R0.
                       ; Adds top halfword of R5 to bottom halfword of R3 and
                       ; writes halved result to bottom halfword of R0.
```

3.5.20 UHSUB16 and UHSUB8

Unsigned Halving Subtract 16 and Unsigned Halving Subtract 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op   Is one of:

        UHSUB16   Performs two unsigned 16-bit integer subtractions, halves the results, and writes the results to the destination register.

        UHSUB8   Performs four unsigned 8-bit integer subtractions, halves the results, and writes the results to the destination register.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
UHSUB16  R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of
                     ; R1 and writes halved result to corresponding halfword in
                     ; R1.
UHSUB8   R4, R0, R5  ; Subtracts bytes of R5 from corresponding byte in R0 and
                     ; writes halved result to corresponding byte in R4.
```

3.5.21 SEL

Select bytes. Selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Syntax

SEL{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

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

3.5.22 USAD8

Unsigned Sum of Absolute Differences.

Syntax

USAD8{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
USAD8 R1, R4, R0    ; Subtracts each byte in R0 from corresponding byte of R4
                    ; adds the differences and writes to R1.
USAD8  R0, R5       ; Subtracts bytes of R5 from corresponding byte in R0,
                    ; adds the differences and writes to R0.
```

3.5.23 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

Syntax

USADA8{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
USADA8 R1, R0, R6      ; Subtracts bytes in R0 from corresponding halfword of R1
                       ; adds differences, adds value of R6, writes to R1.
USADA8  R4, R0, R5, R2 ; Subtracts bytes of R5 from corresponding byte in R0
                       ; adds differences, adds value of R2 writes to R4.
```

3.5.24 USUB16 and USUB8

Unsigned Subtract 16 and Unsigned Subtract 8.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:
	USUB16 Unsigned Subtract 16.
	USUB8 Unsigned Subtract 8.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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';
```

Examples

```
USUB16 R1, R0      ; Subtracts halfwords in R0 from corresponding halfword of R1
                   ; and writes to corresponding halfword in R1.
USUB8  R4, R0, R5  ; Subtracts bytes of R5 from corresponding byte in R0 and
                   ; writes to the corresponding byte in R4.
```

3.6   Multiply and divide instructions

Table 3-9 shows the multiply and divide instructions:

Table 3-9 Multiply and divide instructions

Mnemonic	Brief description	See
MLA	Multiply with Accumulate, 32-bit result	*MUL, MLA, and MLS* on page 3-83
MLS	Multiply and Subtract, 32-bit result	*MUL, MLA, and MLS* on page 3-83
MUL	Multiply, 32-bit result	*MUL, MLA, and MLS* on page 3-83
SDIV	Signed Divide	*SDIV and UDIV* on page 3-103
SMLA[B,T]	Signed Multiply Accumulate (halfwords)	*SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-87
SMLAD, SMLADX	Signed Multiply Accumulate Dual	*SMLAD and SMLADX* on page 3-89
SMLAL	Signed Multiply with Accumulate ($32 \times 32 + 64$), 64-bit result	*UMULL, UMLAL, SMULL, and SMLAL* on page 3-102
SMLAL[B,T]	Signed Multiply Accumulate Long (halfwords)	*SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-91
SMLALD, SMLALDX	Signed Multiply Accumulate Long Dual	*SMLALD, SMLALDX, SMLALBB, SMLALBT, SMLALTB, and SMLALTT* on page 3-91
SMLAW[B|T]	Signed Multiply Accumulate (word by halfword)	*SMLAWB, SMLAWT, SMLABB, SMLABT, SMLATB, and SMLATT* on page 3-87
SMLSD	Signed Multiply Subtract Dual	*SMLSD and SMLSLD* on page 3-93
SMLSLD	Signed Multiply Subtract Long Dual	*SMLSD and SMLSLD* on page 3-93
SMMLA	Signed Most Significant Word Multiply Accumulate	*SMMLA and SMMLS* on page 3-95
SMMLS, SMMLSR	Signed Most Significant Word Multiply Subtract	*SMMLA and SMMLS* on page 3-95
SMUAD, SMUADX	Signed Dual Multiply Add	*SMUAD and SMUSD* on page 3-98
SMUL[B,T]	Signed Multiply (word by halfword)	*SMUL and SMULW* on page 3-100
SMMUL, SMMULR	Signed Most Significant Word Multiply	*SMMUL* on page 3-97
SMULL	Signed Multiply (32×32), 64-bit result	*UMULL, UMLAL, SMULL, and SMLAL* on page 3-102
SMULWB, SMULWT	Signed Multiply (word by halfword)	*SMUL and SMULW* on page 3-100
SMUSD, SMUSDX	Signed Dual Multiply Subtract	*SMUAD and SMUSD* on page 3-98
UDIV	Unsigned Divide	*SDIV and UDIV* on page 3-103
UMAAL	Unsigned Multiply Accumulate Accumulate Long ($32 \times 32 + 32 + 32$), 64-bit result	*UMULL, UMAAL, UMLAL* on page 3-85
UMLAL	Unsigned Multiply with Accumulate ($32 \times 32 + 64$), 64-bit result	*UMULL, UMLAL, SMULL, and SMLAL* on page 3-102
UMULL	Unsigned Multiply (32×32), 64-bit result	*UMULL, UMLAL, SMULL, and SMLAL* on page 3-102

3.6.1 MUL, MLA, and MLS

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

Syntax

```
MUL{S}{cond} {Rd,} Rn, Rm ; Multiply

MLA{cond} Rd, Rn, Rm, Ra  ; Multiply with accumulate

MLS{cond} Rd, Rn, Rm, Ra  ; Multiply with subtract
```

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-20.
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:
- Update the N and Z flags according to the result.
- Do not affect the C and V flags.

Examples

```
MUL    R10, R2, R5    ; Multiply, R10 = R2 × R5
MLA    R10, R2, R1, R5 ; Multiply with accumulate, R10 = (R2 × R1) + R5
MULS   R0, R2, R2     ; Multiply with flag update, R0 = R2 × R2
MULLT  R2, R3, R2     ; Conditionally multiply, R2 = R3 × R2
MLS    R4, R5, R6, R7 ; Multiply with subtract, R4 = R7 - (R5 × R6)
```

3.6.2 UMULL, UMAAL, UMLAL

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

Syntax

op{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

Where:

op	Is one of:

      UMULL   Unsigned Long Multiply.

      UMAAL   Unsigned Long Multiply with Accumulate Accumulate.

      UMLAL   Unsigned Long Multiply, with Accumulate.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
RdHi, *RdLo*	Are the destination registers. For UMAAL and UMLAL they also hold the accumulating value.
Rn	Is the first operand register.
Rm	Is the second operand register.

Operation

These instructions interpret the values from *Rn* and *Rm* as unsigned 32-bit integers.
The UMULL instruction:
- Multiplies the two unsigned integers in the first and second operands.
- Writes the least significant 32 bits of the result in *RdLo.*
- Writes the most significant 32 bits of the result in *RdHi*.

The UMAAL instruction:
- Multiplies the two unsigned 32-bit integers in the first and second operands.
- 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*.
- Writes the lower 32-bits of the result to *RdLo.*

The UMLAL instruction:
- Multiplies the two unsigned integers in the first and second operands.
- Adds the 64-bit result to the 64-bit unsigned integer contained in *RdHi* and *RdLo.*
- 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.

Examples

```
UMULL   R0, R4, R5, R6   ; Multiplies R5 and R6, writes the top 32 bits to R4
                         ; and the bottom 32 bits to R0.
UMAAL   R3, R6, R2, R7   ; Multiplies R2 and R7, adds R6, adds R3, writes the
                         ; top 32 bits to R6, and the bottom 32 bits to R3.
UMLAL   R2, R1, R3, R5   ; Multiplies R5 and R3, adds R1:R2, writes to R1:R2.
```

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

Signed Multiply Accumulate (halfwords).

Syntax

op{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

Where:

op   Is one of:

         SMLAWB   Signed Multiply Accumulate (word by halfword)
                      The bottom halfword, bits [15:0], of *Rm* is used.

         SMLAWT   Signed Multiply Accumulate (word by halfword)
                      The top halfword, bits [31:16] of *Rm* is used.

         SMLABB, SMLABT
                      Signed Multiply Accumulate Long (halfwords)
                      The bottom halfword, bits [15:0], of *Rm* is used.

         SMLATB, SMLATT
                      Signed Multiply Accumulate Long (halfwords)
                      The top halfword, bits [31:16] of *Rm* is used.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
SMLABB  R5, R6, R4, R1  ; Multiplies bottom halfwords of R6 and R4, adds
                        ; R1 and writes to R5.
SMLATB  R5, R6, R4, R1  ; Multiplies top halfword of R6 with bottom halfword
                        ; of R4, adds R1 and writes to R5.
SMLATT  R5, R6, R4, R1  ; Multiplies top halfwords of R6 and R4, adds
                        ; R1 and writes the sum to R5.
SMLABT  R5, R6, R4, R1  ; Multiplies bottom halfword of R6 with top halfword
                        ; of R4, adds R1 and writes to R5.
SMLABT  R4, R3, R2      ; Multiplies bottom halfword of R4 with top halfword of
                        ; R3, adds R2 and writes to R4.
SMLAWB  R10, R2, R5, R3 ; Multiplies R2 with bottom halfword of R5, adds
                        ; R3 to the result and writes top 32-bits to R10.
SMLAWT  R10, R2, R1, R5 ; Multiplies R2 with top halfword of R1, adds R5
                        ; and writes top 32-bits to R10.
```

3.6.4 SMLAD and SMLADX

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

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

Where:

op   Is one of:

        SMLAD   Signed Multiply Accumulate Long Dual.

        SMLADX   Signed Multiply Accumulate Long Dual exchange.

                X specifies which halfword of the source register *Rn* is used as the multiply operand.

                If X is omitted, the multiplications are bottom × bottom and top × top.

                If X is present, the multiplications are bottom × top and top × bottom.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
SMLAD   R10, R2, R1, R5 ; Multiplies two halfword values in R2 with
```

```
                              ; corresponding halfwords in R1, adds R5 and writes to
                              ; R10.
      SMLALDX R0, R2, R4, R6  ; Multiplies top halfword of R2 with bottom halfword
                              ; of R4, multiplies bottom halfword of R2 with top
                              ; halfword of R4, adds R6 and writes to R0.
```

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

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

Syntax

op{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

Where:

op   Is one of:

SMLALBB, SMLALBT

Signed Multiply Accumulate Long (halfwords, B and T).

B and T specify which halfword of the source registers *Rn* and *Rm* are used as the first and second multiply operand:

The bottom halfword, bits [15:0], of *Rn* is used.

SMLALBB: the bottom halfword, bits [15:0], of *Rm* is used.
SMLALBT: the top halfword, bits [31:16], of *Rm* is used.

SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords, B and T).

The top halfword, bits [31:16], of *Rn* is used.

SMLALTB: the bottom halfword, bits [15:0], of *Rm* is used.
SMLALTT: the top halfword, bits [31:16], of *Rm* is used.

SMLALD   Signed Multiply Accumulate Long Dual.

The multiplications are bottom × bottom and top × top.

SMLALDX   Signed Multiply Accumulate Long Dual reversed.

The multiplications are bottom × top and top × bottom.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
SMLAL     R4, R5, R3, R8   ; Multiplies R3 and R8, adds R5:R4 and writes to
                           ; R5:R4.
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.
```

3.6.6   SMLSD and SMLSLD

Signed Multiply Subtract Dual and Signed Multiply Subtract Long Dual.

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*   ; SMLSD

op{X}{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*   ; SMLSLD

Where:

op	Is one of:

	SMLSD	Signed Multiply Subtract Dual.
	SMLSDX	Signed Multiply Subtract Dual reversed.
	SMLSLD	Signed Multiply Subtract Long Dual.
	SMLSLDX	Signed Multiply Subtract Long Dual reversed.

If X is present, the multiplications are bottom × top and top × bottom.
If the X is omitted, the multiplications are bottom × bottom and top × top.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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 Thumb instruction set, these instructions do not affect the condition code flags.

Examples

```
SMLSD     R0, R4, R5, R6   ; Multiplies bottom halfword of R4 with bottom
                           ; halfword of R5, multiplies top halfword of R4
                           ; with top halfword of R5, subtracts second from
                           ; first, adds R6, writes to R0.
SMLSDX    R1, R3, R2, R0   ; Multiplies bottom halfword of R3 with top
                           ; halfword of R2, multiplies top halfword of R3
                           ; with bottom halfword of R2, subtracts second from
                           ; first, adds R0, writes to R1.
SMLSLD    R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with bottom
                           ; halfword of R2, multiplies top halfword of R6
                           ; with top halfword of R2, subtracts second from
                           ; first, adds R6:R3, writes to R6:R3.
SMLSLDX   R3, R6, R2, R7   ; Multiplies bottom halfword of R6 with top
                           ; halfword of R2, multiplies top halfword of R6
                           ; with bottom halfword of R2, subtracts second from
                           ; first, adds R6:R3, writes to R6:R3.
```

3.6.7 SMMLA and SMMLS

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

Syntax

op{R}{*cond*} *Rd*, *Rn*, *Rm*, *Ra*

Where:

op	Is one of:
	SMMLA Signed Most Significant Word Multiply Accumulate.
	SMMLS Signed Most Significant Word Multiply Subtract.
R	If R is present, the result is rounded instead of being truncated. In this case the constant 0x80000000 is added to the product before the top halfword is extracted.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SMMLA    R0, R4, R5, R6    ; Multiplies R4 and R5, extracts top 32 bits, adds
                           ; R6, truncates and writes to R0.
SMMLAR   R6, R2, R1, R4    ; Multiplies R2 and R1, extracts top 32 bits, adds
                           ; 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.
```

3.6.8 SMMUL

Signed Most Significant Word Multiply.

Syntax

op{R}{*cond*} *Rd*, *Rn*, *Rm*

Where:

op	Is one of:
	SMMUL Signed Most Significant Word Multiply
R	If R is present, the result is rounded instead of being truncated. In this case the constant 0x80000000 is added to the product before the top halfword is extracted.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SMULL    R0, R4, R5      ; Multiplies R4 and R5, truncates top 32 bits
                         ; and writes to R0.
SMULLR   R6, R2          ; Multiplies R6 and R2, rounds the top 32 bits
                         ; and writes to R6.
```

3.6.9   SMUAD and SMUSD

Signed Dual Multiply Add and Signed Dual Multiply Subtract.

Syntax

op{X}{*cond*} *Rd*, *Rn*, *Rm*

Where:

op	Is one of:

	SMUAD	Signed Dual Multiply Add.
	SMUADX	Signed Dual Multiply Add reversed.
	SMUSD	Signed Dual Multiply Subtract.
	SMUSDX	Signed Dual Multiply Subtract reversed.

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. See *Conditional execution* on page 3-20.
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.

Examples

```
SMUAD    R0, R4, R5    ; Multiplies bottom halfword of R4 with the bottom
                       ; halfword of R5, adds multiplication of top halfword
                       ; of R4 with top halfword of R5, writes to R0.
SMUADX   R3, R7, R4    ; Multiplies bottom halfword of R7 with top halfword
                       ; of R4, adds multiplication of top halfword of R7
                       ; with bottom halfword of R4, writes to R3.
SMUSD    R3, R6, R2    ; Multiplies bottom halfword of R4 with bottom halfword
                       ; of R6, subtracts multiplication of top halfword of R6
                       ; with top halfword of R3, writes to R3.
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.
```

3.6.10   SMUL and SMULW

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

Syntax

op{*XY*}{*cond*} *Rd,Rn, Rm*   ; SMUL

op{*Y*}{*cond*} *Rd. Rn, Rm*   ; SMULW

For SMUL{*XY*} only:

op   Is one of SMULBB, SMULBT, SMULTB, SMULTT:

SMUL{*XY*} Signed Multiply (halfwords).

X and *Y* specify which halfword of the source registers *Rn* and *Rm* is used as the first and second multiply operand.
If *X* is B, then the bottom halfword, bits [15:0] of *Rn* is used.
If *X* is T, then the top halfword, bits [31:16] of *Rn* is used.If *Y* is B, then the bottom halfword, bits [15:0], of *Rm* is used.
If *Y* is T, then the top halfword, bits [31:16], of *Rm* is used.

SMULW{*Y*} Signed Multiply (word by halfword).

Y specifies which halfword of the source register *Rm* is used as the second multiply operand.
If *Y* is B, then the bottom halfword (bits [15:0]) of *Rm* is used.
If *Y* is T, then the top halfword (bits [31:16]) of *Rm* is used.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
SMULBT    R0, R4, R5      ; Multiplies the bottom halfword of R4 with the
                          ; top halfword of R5, multiplies results and
                          ; writes to R0.
SMULBB    R0, R4, R5      ; Multiplies the bottom halfword of R4 with the
                          ; bottom halfword of R5, multiplies results and
                          ; writes to R0.
SMULTT    R0, R4, R5      ; Multiplies the top halfword of R4 with the top
                          ; 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.
```

3.6.11 UMULL, UMLAL, SMULL, and SMLAL

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

Syntax

op{*cond*} *RdLo*, *RdHi*, *Rn*, *Rm*

Where:

op	Is one of:

	UMULL	Unsigned Multiply Long.
	UMLAL	Unsigned Multiply, with Accumulate Long.
	SMULL	Signed Multiply Long.
	SMLAL	Signed Multiply, with Accumulate Long.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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 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.

Examples

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

3.6.12 SDIV and UDIV

Signed Divide and Unsigned Divide.

Syntax

SDIV{*cond*} {*Rd*,} *Rn*, *Rm*

UDIV{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

```
SDIV  R0, R2, R4  ; Signed divide, R0 = R2/R4
UDIV  R8, R8, R1  ; Unsigned divide, R8 = R8/R1
```

3.7 Saturating instructions

Table 3-10 shows the saturating instructions:

Table 3-10 Saturating instructions

Mnemonic	Brief description	See
SSAT	Signed Saturate	*SSAT and USAT* on page 3-105
SSAT16	Signed Saturate Halfword	*SSAT16 and USAT16* on page 3-106
USAT	Unsigned Saturate	*SSAT and USAT* on page 3-105
USAT16	Unsigned Saturate Halfword	*SSAT16 and USAT16* on page 3-106
QADD	Saturating Add	*QADD and QSUB* on page 3-107
QSUB	Saturating Subtract	*QADD and QSUB* on page 3-107
QSUB16	Saturating Subtract 16	*QADD and QSUB* on page 3-107
QASX	Saturating Add and Subtract with Exchange	*QASX and QSAX* on page 3-109
QSAX	Saturating Subtract and Add with Exchange	*QASX and QSAX* on page 3-109
QDADD	Saturating Double and Add	*QDADD and QDSUB* on page 3-111
QDSUB	Saturating Double and Subtract	*QDADD and QDSUB* on page 3-111
UQADD16	Unsigned Saturating Add 16	*UQADD and UQSUB* on page 3-114
UQADD8	Unsigned Saturating Add 8	*UQADD and UQSUB* on page 3-114
UQASX	Unsigned Saturating Add and Subtract with Exchange	*UQASX and UQSAX* on page 3-112
UQSAX	Unsigned Saturating Subtract and Add with Exchange	*UQASX and UQSAX* on page 3-112
UQSUB16	Unsigned Saturating Subtract 16	*UQADD and UQSUB* on page 3-114
UQSUB8	Unsigned Saturating Subtract 8	*UQADD and UQSUB* on page 3-114

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, see *MSR* on page 3-179.

To read the state of the Q flag, use the MRS instruction, see *MRS* on page 3-178.

3.7.1 SSAT and USAT

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

Syntax

op{*cond*} *Rd*, #*n*, *Rm* {, *shift* #s}

Where:

op	Is one of:
	SSAT Saturates a signed value to a signed range.
	USAT Saturates a signed value to an unsigned range.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
    SSAT    R7, #16, R7, LSL #4  ; Logical shift left value in R7 by 4, then
                                 ; saturate it as a signed 16-bit value and
                                 ; write it back to R7.
    USATNE  R0, #7, R5           ; Conditionally saturate value in R5 as an
                                 ; unsigned 7 bit value and write it to R0.
```

3.7.2   SSAT16 and USAT16

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

Syntax

op{*cond*} *Rd*, *#n*, *Rm*

Where:

op	Is one of:
	SSAT16 Saturates a signed halfword value to a signed range.
	USAT16 Saturates a signed halfword value to an unsigned range.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SSAT16    R7, #9, R2      ; Saturates the top and bottom highwords of R2
                          ; as 9-bit values, writes to corresponding halfword
                          ; of R7.
USAT16NE  R0, #13, R5     ; Conditionally saturates the top and bottom
                          ; halfwords of R5 as 13-bit values, writes to
                          ; corresponding halfword of R0.
```

3.7.3 QADD and QSUB

Saturating Add and Saturating Subtract, signed.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	QADD	Saturating 32-bit add.
	QADD8	Saturating four 8-bit integer additions.
	QADD16	Saturating two 16-bit integer additions.
	QSUB	Saturating 32-bit subtraction.
	QSUB8	Saturating four 8-bit integer subtraction.
	QSUB16	Saturating two 16-bit integer subtraction.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.	
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, see *MSR* on page 3-179.

To read the state of the Q flag, use the MRS instruction, see *MRS* on page 3-178.

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.

Examples

```
QADD16   R7, R4, R2   ; Adds halfwords of R4 with corresponding halfword of
                      ; R2, saturates to 16 bits and writes to corresponding
                      ; halfword of R7.
QADD8    R3, R1, R6   ; Adds bytes of R1 to the corresponding bytes of R6,
                      ; saturates to 8 bits and writes to corresponding byte of
                      ; R3.
QSUB16   R4, R2, R3   ; Subtracts halfwords of R3 from corresponding halfword
                      ; 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.
```

3.7.4 QASX and QSAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op   Is one of:
QASX   Add and Subtract with Exchange and Saturate.
QSAX   Subtract and Add with Exchange and Saturate.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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 $-2^{15} \leq x \leq 2^{15} - 1$, 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 $-2^{15} \leq x \leq 2^{15} - 1$, 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 $-2^{15} \leq x \leq 2^{15} - 1$, 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 $-2^{15} \leq x \leq 2^{15} - 1$, 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.

Examples

```
QASX    R7, R4, R2   ; Adds top halfword of R4 to bottom halfword of R2,
                     ; saturates to 16 bits, writes to top halfword of R7
                     ; Subtracts top highword of R2 from bottom halfword of
                     ; R4, saturates to 16 bits and writes to bottom halfword
                     ; of R7
QSAX    R0, R3, R5   ; Subtracts bottom halfword of R5 from top halfword of
                     ; R3, saturates to 16 bits, writes to top halfword of R0
                     ; Adds bottom halfword of R3 to top halfword of R5,
                     ; saturates to 16 bits, writes to bottom halfword of R0.
```

3.7.5   QDADD and QDSUB

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

Syntax

op{*cond*} {*Rd*}, *Rm*, *Rn*

Where:

op	Is one of:
	QDADD Saturating Double and Add.
	QDSUB Saturating Double and Subtract.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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 $-2^{31} \le x \le 2^{31} - 1$. 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.

Examples

```
QDADD    R7, R4, R2        ; Doubles and saturates R4 to 32 bits, adds R2,
                           ; saturates to 32 bits, writes to R7
QDSUB    R0, R3, R5        ; Subtracts R3 doubled and saturated to 32 bits
                           ; from R5, saturates to 32 bits, writes to R0.
```

3.7.6 UQASX and UQSAX

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

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

type	Is one of:
	UQASX Add and Subtract with Exchange and Saturate.
	UQSAX Subtract and Add with Exchange and Saturate.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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 $0 \le x \le 216 - 1$, 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 $0 \le x \le 216 - 1$, 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.

Examples

```
UQASX   R7, R4, R2  ; Adds top halfword of R4 with bottom halfword of R2,
                    ; saturates to 16 bits, writes to top halfword of R7
                    ; Subtracts top halfword of R2 from bottom halfword of
                    ; R4, saturates to 16 bits, writes to bottom halfword of R7
UQSAX   R0, R3, R5  ; Subtracts bottom halfword of R5 from top halfword of R3,
                    ; saturates to 16 bits, writes to top halfword of R0
                    ; Adds bottom halfword of R4 to top halfword of R5
                    ; saturates to 16 bits, writes to bottom halfword of R0.
```

3.7.7   UQADD and UQSUB

Saturating Add and Saturating Subtract Unsigned.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm*

Where:

op	Is one of:	
	UQADD8	Saturating four unsigned 8-bit integer additions.
	UQADD16	Saturating two unsigned 16-bit integer additions.
	UQSUB8	Saturating four unsigned 8-bit integer subtractions.
	UQSUB16	Saturating two unsigned 16-bit integer subtractions.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.	
Rd	Is the destination register. If *Rd* is omitted, the destination register is *Rn*.	
Rn, Rm	Are registers holding the first and second operands.	

Operation

These instructions add or subtract two or four values and then writes an unsigned saturated value in the destination register.

The UQADD16 instruction:
- Adds the respective top and bottom halfwords of the first and second operands.
- Saturates the result of the additions for each halfword in the destination register to the unsigned range $0 \leq x \leq 2^{16}-1$, where x is 16.

The UQADD8 instruction:
- Adds each respective byte of the first and second operands.
- Saturates the result of the addition for each byte in the destination register to the unsigned range $0 \leq x \leq 2^{8}-1$, where x is 8.

The UQSUB16 instruction:
- Subtracts both halfwords of the second operand from the respective halfwords of the first operand.
- Saturates the result of the differences in the destination register to the unsigned range $0 \leq x \leq 2^{16}-1$, where x is 16.

The UQSUB8 instructions:
- Subtracts the respective bytes of the second operand from the respective bytes of the first operand.
- Saturates the results of the differences for each byte in the destination register to the unsigned range $0 \leq x \leq 2^{8}-1$, where x is 8.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Examples

```
UQADD16  R7, R4, R2   ; Adds halfwords in R4 to corresponding halfword in R2,
                      ; saturates to 16 bits, writes to corresponding halfword
                      ; of R7
UQADD8   R4, R2, R5   ; Adds bytes of R2 to corresponding byte of R5, saturates
                      ; to 8 bits, writes to corresponding bytes of R4
UQSUB16  R6, R3, R0   ; Subtracts halfwords in R0 from corresponding halfword
                      ; in R3, saturates to 16 bits, writes to corresponding
                      ; halfword in R6
UQSUB8   R1, R5, R6   ; Subtracts bytes in R6 from corresponding byte of R5,
                      ; saturates to 8 bits, writes to corresponding byte of R1.
```

3.8 Packing and unpacking instructions

Table 3-11shows the instructions that operate on packing and unpacking data:

Table 3-11 Packing and unpacking instructions

Mnemonic	Brief description	See
PKH	Pack Halfword	*PKHBT and PKHTB* on page 3-117
SXTAB	Extend 8 bits to 32 and add	*SXTA and UXTA* on page 3-121
SXTAB16	Dual extend 8 bits to 16 and add	*SXTA and UXTA* on page 3-121
SXTAH	Extend 16 bits to 32 and add	*SXTA and UXTA* on page 3-121
SXTB	Sign extend a byte	*SXT and UXT* on page 3-126
SXTB16	Dual extend 8 bits to 16 and add	*SXT and UXT* on page 3-126
SXTH	Sign extend a halfword	*SXT and UXT* on page 3-126
UXTAB	Extend 8 bits to 32 and add	*SXTA and UXTA* on page 3-121
UXTAB16	Dual extend 8 bits to 16 and add	*SXTA and UXTA* on page 3-121
UXTAH	Extend 16 bits to 32 and add	*SXTA and UXTA* on page 3-121
UXTB	Zero extend a byte	*SXT and UXT* on page 3-126
UXTB16	Dual zero extend 8 bits to 16 and add	*SXT and UXT* on page 3-126
UXTH	Zero extend a halfword	*SXT and UXT* on page 3-126

3.8.1 PKHBT and PKHTB

Pack Halfword.

Syntax

```
op{cond} {Rd}, Rn, Rm {, LSL #imm}      ;PKHBT
op{cond} {Rd}, Rn, Rm {, ASR #imm}      ;PKHTB
```

Where:

op	Is one of:
	PKHBT Pack Halfword, bottom and top with shift.
	PKHTB Pack Halfword, top and bottom with shift.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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

LSL   A left shift with a shift length from 1 to 31, 0 means no shift.

For PKHTB:

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.

Examples

```
PKHBT   R3, R4, R5 LSL #0  ; Writes bottom halfword of R4 to bottom halfword of
                           ; R3, writes top halfword of R5, unshifted, to top
                           ; halfword of R3
PKHTB   R4, R0, R2 ASR #1  ; Writes R2 shifted right by 1 bit to bottom halfword
                           ; of R4, and writes top halfword of R0 to top
                           ; halfword of R4.
```

3.8.2 SXT and UXT

Sign extend and Zero extend.

Syntax

op{*cond*} *Rd*, *Rn* {, ROR #*n*}   ; SXTB; SXTH; UXTB; UXTH

op{*cond*} {*Rd*}, *Rn* {, ROR #*n*}   ; SXTB16; UXTB16

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. See *Conditional execution* on page 3-20.

Rd   Is the destination register. If *Rd* is omitted, the destination register is *Rn*.

Rn   Is the register holding the value to 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:
   - 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.

Examples

```
SXTH  R4, R6, ROR #16  ; Rotates R6 right by 16 bits, obtains bottom halfword of
                       ; of result, sign extends to 32 bits and writes to R4
UXTB  R3, R10          ; Extracts lowest byte of value in R10, zero extends, and
                       ; writes to R3.
```

3.8.3   SXTA and UXTA

Signed and Unsigned Extend and Add.

Syntax

op{*cond*} {*Rd*,} *Rn*, *Rm* {, ROR #*n*}

Where:

op	Is one of:

	SXTAB	Sign extends an 8-bit value to a 32-bit value and add.
	SXTAH	Sign extends a 16-bit value to a 32-bit value and add.
	SXTAB16	Sign extends two 8-bit values to two 16-bit values and add.
	UXTAB	Zero extends an 8-bit value to a 32-bit value and add.
	UXTAH	Zero extends a 16-bit value to a 32-bit value and add.
	UXTAB16	Zero extends two 8-bit values to two 16-bit values and add.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SXTAH  R4, R8, R6, ROR #16 ; Rotates R6 right by 16 bits, obtains bottom
                           ; halfword, sign extends to 32 bits, adds R8,and
                           ; writes to R4
UXTAB  R3, R4, R10         ; Extracts bottom byte of R10 and zero extends to 32
                           ; bits, adds R4, and writes to R3.
```

3.9   Bit field instructions

Table 3-12 shows the instructions that operate on adjacent sets of bits in registers or bit fields:

Table 3-12 Bit field instructions

Mnemonic	Brief description	See
BFC	Bit Field Clear	*BFC and BFI* on page 3-124
BFI	Bit Field Insert	*BFC and BFI* on page 3-124
SBFX	Signed Bit Field Extract	*SBFX and UBFX* on page 3-125
SXTB	Sign extend a byte	*SXT and UXT* on page 3-126
SXTH	Sign extend a halfword	*SXT and UXT* on page 3-126
UBFX	Unsigned Bit Field Extract	*SBFX and UBFX* on page 3-125
UXTB	Zero extend a byte	*SXT and UXT* on page 3-126
UXTH	Zero extend a halfword	*SXT and UXT* on page 3-126

3.9.1　BFC and BFI

Bit Field Clear and Bit Field Insert.

Syntax

```
BFC{cond} Rd, #lsb, #width
```

```
BFI{cond} Rd, Rn, #lsb, #width
```

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
BFC   R4, #8, #12      ; Clear bit 8 to bit 19 (12 bits) of R4 to 0
BFI   R9, R2, #8, #12  ; Replace bit 8 to bit 19 (12 bits) of R9 with
                       ; bit 0 to bit 11 from R2.
```

3.9.2   SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Syntax

SBFX{*cond*} *Rd*, *Rn*, #*lsb*, #*width*

UBFX{*cond*} *Rd*, *Rn*, #*lsb*, #*width*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

```
SBFX  R0, R1, #20, #4  ; Extract bit 20 to bit 23 (4 bits) from R1 and sign
                       ; extend to 32 bits and then write the result to R0.
UBFX  R8, R11, #9, #10 ; Extract bit 9 to bit 18 (10 bits) from R11 and zero
                       ; extend to 32 bits and then write the result to R8.
```

3.9.3   SXT and UXT

Sign extend and Zero extend.

Syntax

SXT*extend*{*cond*} *Rd*, *Rn* {, ROR #*n*}

UXT*extend*{*cond*} *Rd*, *Rn* {, ROR #*n*}

Where:

extend	Is one of:
	B Extends an 8-bit value to a 32-bit value.
	H Extends a 16-bit value to a 32-bit value.
cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```
SXTH  R4, R6, ROR #16  ; Rotate R6 right by 16 bits, then obtain the lower
                       ; halfword of the result and then sign extend to
                       ; 32 bits and write the result to R4.
UXTB  R3, R10          ; Extract lowest byte of the value in R10 and zero
                       ; extend it, and write the result to R3.
```

3.10 Branch and control instructions

Table 3-13 shows the branch and control instructions:

Table 3-13 Branch and control instructions

Mnemonic	Brief description	See
B	Branch	*B, BL, BX, and BLX* on page 3-128
BL	Branch with Link	*B, BL, BX, and BLX* on page 3-128
BLX	Branch indirect with Link	*B, BL, BX, and BLX* on page 3-128
BX	Branch indirect	*B, BL, BX, and BLX* on page 3-128
CBNZ	Compare and Branch if Non Zero	*CBZ and CBNZ* on page 3-130
CBZ	Compare and Branch if Zero	*CBZ and CBNZ* on page 3-130
IT	If-Then	*IT* on page 3-131
TBB	Table Branch Byte	*TBB and TBH* on page 3-133
TBH	Table Branch Halfword	*TBB and TBH* on page 3-133

3.10.1 B, BL, BX, and BLX

Branch instructions.

Syntax

B{*cond*} *label*

BL{*cond*} *label*

BX{*cond*} *Rm*

BLX{*cond*} *Rm*

Where:

B   Is branch (immediate).

BL   Is branch with link (immediate).

BX   Is branch indirect (register).

BLX   Is branch indirect with link (register).

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

label   Is a PC-relative expression. See *PC-relative expressions* on page 3-19.

Rm   Is a register that indicates an address to branch to. Bit[0] of the value in *Rm* must be 1, but the address to branch to is created by changing bit[0] to 0.

Operation

All these instructions cause a branch to *label*, or to the address indicated in *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.

B*cond* label is the only conditional instruction that can be either inside or outside an IT block. All other branch instructions must be conditional inside an IT block, and must be unconditional outside the IT block, see *IT* on page 3-131.

Table 3-14 shows the ranges for the various branch instructions.

Table 3-14 Branch ranges

Instruction	Branch range
B label	−16 MB to +16 MB
B*cond* label (outside IT block)	−1 MB to +1 MB
B*cond* label (inside IT block)	−16 MB to +16 MB
BL{*cond*} label	−16 MB to +16 MB
BX{*cond*} Rm	Any value in register
BLX{*cond*} Rm	Any value in register

> ─── **Note** ───
>
> You might have to use the `.W` suffix to get the maximum branch range. See *Instruction width selection* on page 3-23.

Restrictions

The restrictions are:

*   Do not use PC in the `BLX` instruction.

*   For `BX` and `BLX`, bit[0] of *Rm* must be 1 for correct execution but a branch occurs to the target address created by changing bit[0] to 0.

*   When any of these instructions is inside an IT block, it must be the last instruction of the IT block.

> ─── **Note** ───
>
> B*cond* is the only conditional instruction that is not required to be inside an IT block. However, it has a longer branch range when it is inside an IT block.

Condition flags

These instructions do not change the flags.

Examples

```
B     loopA  ; Branch to loopA
BLE   ng     ; Conditionally branch to label ng
B.W   target ; Branch to target within 16MB range
BEQ   target ; Conditionally branch to target
BEQ.W target ; Conditionally branch to target within 1MB
BL    funC   ; Branch with link (Call) to function funC, return address
             ; stored in LR
BX    LR     ; Return from function call
BXNE  R0     ; Conditionally branch to address stored in R0
BLX   R0     ; Branch with link and exchange (Call) to a address stored
             ; in R0.
```

3.10.2   CBZ and CBNZ

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

Syntax

op{*cond*} *Rn*, *label*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.
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.

Examples

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

3.10.3 IT

If-Then condition instruction.

Syntax

IT{*x*{*y*{*z*}}} *cond*

Where:

x	specifies the condition switch for the second instruction in the IT block.
y	Specifies the condition switch for the third instruction in the IT block.
z	Specifies the condition switch for the fourth instruction in the IT block.
cond	Specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:

T	Then. Applies the condition *cond* to the instruction.
E	Else. Applies the inverse condition of *cond* to the instruction.

——— **Note** ———

It is possible to use AL (the *always* condition) for *cond* in an IT instruction. If this is done, all of the instructions in the IT block must be unconditional, and each of *x*, *y*, and *z* must be T or omitted but not E.

Operation

The IT instruction makes up to four following instructions conditional. The conditions can be all the same, or some of them can be the logical inverse of the others. The conditional instructions following the IT instruction form the *IT block*.

The instructions in the IT block, including any branches, must specify the condition in the {*cond*} part of their syntax.

——— **Note** ———

Your assembler might be able to generate the required IT instructions for conditional instructions automatically, so that you do not have to write them yourself. See your assembler documentation for details.

A BKPT instruction in an IT block is always executed, even if its condition fails.

Exceptions can be taken between an IT instruction and the corresponding IT block, or within an IT block. Such an exception results in entry to the appropriate exception handler, with suitable return information in LR and stacked PSR.

Instructions designed for use for exception returns can be used as normal to return from the exception, and execution of the IT block resumes correctly. This is the only way that a PC-modifying instruction is permitted to branch to an instruction in an IT block.

Restrictions

The following instructions are not permitted in an IT block:

*   IT.
*   CBZ and CBNZ.
*   CPSID and CPSIE.

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. These are:
  — `ADD PC, PC,` *Rm.*
  — `MOV PC,` *Rm.*
  — `B, BL, BX, BLX.`
  — Any `LDM, LDR,` or `POP` instruction that writes to the PC.
  — `TBB` and `TBH`.

- Do not branch to any instruction inside an IT block, except when returning from an exception handler.

- All conditional instructions except B*cond* must be inside an IT block. B*cond* can be either outside or inside an IT block but has a larger branch range if it is inside one.

- Each instruction inside the IT block must specify a condition code suffix that is either the same or logical inverse as for the other instructions in the block.

Note

Your assembler might place extra restrictions on the use of IT blocks, such as prohibiting the use of assembler directives within them.

Condition flags

This instruction does not change the flags.

Example

```
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
```

3.10.4   TBB and TBH

Table Branch Byte and Table Branch Halfword.

Syntax

TBB [*Rn*, *Rm*]

TBH [*Rn*, *Rm*, LSL #1]

Where:

Rn	Is the register containing the address of the table of branch lengths.
	If *Rn* is PC, then the address of the table is the address of the byte immediately following the TBB or TBH instruction.
Rm	Is the index register. This contains an index into the table. For halfword tables, LSL #1 doubles the value in *Rm* to form the right offset into the table.

Operation

These instructions cause a PC-relative forward branch using a table of single byte offsets for TBB, or halfword offsets for TBH. *Rn* provides a pointer to the table, and *Rm* supplies an index into the table. For TBB the branch offset is the unsigned value of the byte returned from the table, and for TBH the branch offset is twice the unsigned value of the halfword returned from the table. The branch occurs to the address at that offset from the address of the byte immediately after the TBB or TBH instruction.

Restrictions

The restrictions are:
- *Rn* must not be SP.
- *Rm* must not be SP and must not be PC.
- When any of these instructions is used inside an IT block, it must be the last instruction of the IT block.

Condition flags

These instructions do not change the flags.

Examples

```
     ADR.W  R0, BranchTable_Byte
     TBB    [R0, R1]           ; R1 is the index, R0 is the base address of the
                               ; branch table
Case1
; an instruction sequence follows
Case2
; an instruction sequence follows
Case3
; an instruction sequence follows
BranchTable_Byte
     DCB    0                  ; Case1 offset calculation
     DCB    ((Case2-Case1)/2)  ; Case2 offset calculation
     DCB    ((Case3-Case1)/2)  ; Case3 offset calculation




     TBH    [PC, R1, LSL #1]   ; R1 is the index, PC is used as base of the
                               ; branch table
BranchTable_H
     DCW    ((CaseA - BranchTable_H)/2)  ; CaseA offset calculation
     DCW    ((CaseB - BranchTable_H)/2)  ; CaseB offset calculation
     DCW    ((CaseC - BranchTable_H)/2)  ; CaseC offset calculation

CaseA
; an instruction sequence follows
CaseB
; an instruction sequence follows
CaseC
; an instruction sequence follows
```

3.11 Floating-point instructions

This section provides the instruction set that the single-precision and double-precision FPU uses.

Table 3-15 shows the floating-point instructions.

— **Note** —

These instructions are only available if the FPU is included, and enabled, in the system. See *Enabling the FPU* on page 4-59 for information about enabling the floating-point unit.

Table 3-15 Floating-point instructions

Mnemonic	Brief description	See
VABS	Floating-point Absolute	*VABS* on page 3-137
VADD	Floating-point Add	*VADD* on page 3-138
VCMP	Compare two floating-point registers, or one floating-point register and zero	*VCMP and VCMPE* on page 3-139
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	*VCMP and VCMPE* on page 3-139
VCVT	Convert between floating-point and integer	*VCVT and VCVTR between floating-point and integer* on page 3-140
VCVT	Convert between floating-point and fixed point	*VCVT between floating-point and fixed-point* on page 3-141
VCVTR	Convert between floating-point and integer with rounding	*VCVT and VCVTR between floating-point and integer* on page 3-140
VCVTB	Converts half-precision value to single-precision	*VCVTB and VCVTT* on page 3-142
VCVTT	Converts single-precision register to half-precision	*VCVTB and VCVTT* on page 3-142
VDIV	Floating-point Divide	*VDIV* on page 3-143
VFMA	Floating-point Fused Multiply Accumulate	*VFMA and VFMS* on page 3-144
VFNMA	Floating-point Fused Negate Multiply Accumulate	*VFNMA and VFNMS* on page 3-145
VFMS	Floating-point Fused Multiply Subtract	*VFMA and VFMS* on page 3-144
VFNMS	Floating-point Fused Negate Multiply Subtract	*VFNMA and VFNMS* on page 3-145
VLDM	Load Multiple extension registers	*VLDM* on page 3-146
VLDR	Loads an extension register from memory	*VLDR* on page 3-147
VMLA	Floating-point Multiply Accumulate	*VMLA and VMLS* on page 3-148
VMLS	Floating-point Multiply Subtract	*VMLA and VMLS* on page 3-148
VMOV	Floating-point Move Immediate	*VMOV Immediate* on page 3-149
VMOV	Floating-point Move Register	*VMOV Register* on page 3-150
VMOV	Copy ARM core register to single-precision	*VMOV ARM Core register to single-precision* on page 3-152

Table 3-15 Floating-point instructions (continued)

Mnemonic	Brief description	See
VMOV	Copy 2 ARM core registers to 2 single-precision	*VMOV two ARM Core registers to two single-precision registers* on page 3-153
VMOV	Copies between ARM core register to scalar	*VMOV ARM Core register to scalar* on page 3-155
VMOV	Copies between Scalar to ARM core register	*VMOV Scalar to ARM Core register* on page 3-151
VMRS	Move to ARM core register from floating-point System Register	*VMRS* on page 3-156
VMSR	Move to floating-point System Register from ARM Core register	*VMSR* on page 3-157
VMUL	Multiply floating-point	*VMUL* on page 3-158
VNEG	Floating-point negate	*VNEG* on page 3-159
VNMLA	Floating-point multiply and add	*VNMLA, VNMLS and VNMUL* on page 3-160
VNMLS	Floating-point multiply and subtract	*VNMLA, VNMLS and VNMUL* on page 3-160
VNMUL	Floating-point multiply	*VNMLA, VNMLS and VNMUL* on page 3-160
VPOP	Pop extension registers	*VPOP* on page 3-161
VPUSH	Push extension registers	*VPUSH* on page 3-162
VSQRT	Floating-point square root	*VSQRT* on page 3-163
VSTM	Store Multiple extension registers	*VSTM* on page 3-164
VSTR	Stores an extension register to memory	*VSTR* on page 3-165
VSUB	Floating-point Subtract	*VSUB* on page 3-166
VSEL	Select register, alternative to a pair of conditional VMOV	*VSEL* on page 3-167
VMAXNM, VMINNM	Maximum, Minimum with IEEE754-2008 NaN handling	*VMAXNM and VMINNM* on page 3-168
VCVTA, VCVTN, VCVTP, VCVTM	Float to integer conversion with directed rounding	*VCVTA, VCVTN, VCVTP and VCVTM* on page 3-169
VRINTR, VRINTX	Float to integer (in floating-point format) conversion	*VRINTR and VRINTX* on page 3-170
VRINTA, VRINTN, VRINTP, VRINTM	Float to integer (in floating-point format) conversion with directed rounding	*VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ* on page 3-171

3.11.1 VABS

Floating-point Absolute.

Syntax

```
VABS{cond}.F<32|64> <Sd|Dd>, <Sm|Dm>
```

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

`<Sd|Dd>, <Sm|Dm>`

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.

Examples

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

3.11.2 VADD

Floating-point Add.

Syntax

VADD{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sn|Dn>, <Sm|Dm>

            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.

Restrictions

There are no restrictions.

Condition flags

This instruction does not change the flags.

Examples

VADD.F32 S4, S6, S7

3.11.3   VCMP and VCMPE

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

Syntax

VCMP{*E*}{*cond*}.F<32|64> *<Sd|Dd>*, *<Sm|#0.0>*

VCMP{*E*}{*cond*}.F<32|64> *<Sd|Dd>*, #0.0

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
E	If present, any NaN operand causes an Invalid Operation exception. Otherwise, only a signaling NaN causes the exception.
<Sd|Dd>	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, see *VMRS* on page 3-156.

Examples

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

3.11.4 VCVT and VCVTR between floating-point and integer

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

Syntax

VCVT{R}{*cond*}.*Tm*.F<32|64> *<Sd>, <Sm|Dm>*

VCVT{*cond*}.F<32|64>.*Tm* *<Sd|Dd>, <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. See *Conditional execution* on page 3-20.
Tm	Is the data type for the operand. It must be one of:

  • S32 signed 32-bit value.
  • U32 unsigned 32-bit value.

<Sd|Dd>, <Sm|Dm>

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.

3.11.5   VCVT between floating-point and fixed-point

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

Syntax

VCVT{*cond*}.*Td*.F<32|64> *<Sd|Dd>*, *<Sd|Dd>*, #*fbits*

VCVT{*cond*}.F<32|64>.*Td <Sd|Dd>*, *<Sd|Dd>*, #*fbits*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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|Dd>	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.

3.11.6   VCVTB and VCVTT

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

Syntax

VCVT{*y*}{*cond*}.F<32|64>.F16 *<Sd|Dd>*, *Sm*

VCVT{*y*}{*cond*}.F16.F<32|64> *Sd*, *<Sm|Dm>*

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. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination register.

<Sm|Dm>   Is the operand register.

Operation

This instruction with the .F16.F<32|64> suffix:

1. Converts the half-precision value in the top or bottom half of a single-precision register to single-precision or double-precision value.
2. Writes the result to a single-precision or double-precision register.

This instruction with the .F<32|64>.F16 suffix:

1. Converts the value in a double-precision or 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.

3.11.7   VDIV

Divides floating-point values.

Syntax

VDIV{*cond*}.F<32|64> {<*Sd*|*Dd*>,} <*Sn*|*Dn*>, <*Sm*|*Dm*>

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination register.

<Sn|Dn>, <Sm|Dm>

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.

3.11.8   VFMA and VFMS

Floating-point Fused Multiply Accumulate and Subtract.

Syntax

VFMA{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

VFMS{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination register.

<Sn|Dn>, <Sm|Dm>

            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.

3.11.9 VFNMA and VFNMS

Floating-point Fused Negate Multiply Accumulate and Subtract.

Syntax

VFNMA{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

VFNMS{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination register.

<Sn|Dn>, <Sm|Dm>

                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.

3.11.10 VLDM

Floating-point Load Multiple.

Syntax

VLDM{*mode*}{*cond*}{.*size*} *Rn*{!}, *list*

Where:

mode	Is the addressing mode:

IA   `Increment after`. The consecutive addresses start at the address specified in *Rn*.

DB   `Decrement before`. The consecutive addresses end before the address specified in *Rn*.

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
size	Is an optional data size specifier.
Rn	Is the base register. The SP can be used.
!	Is the command to the instruction to write a modified value back to *Rn*. This is required if mode == DB, and is optional if mode == IA.
list	Is the list of extension registers to be loaded, as a list of consecutively numbered doubleword or singleword registers, separated by commas and surrounded by brackets.

Operation

This instruction loads multiple extension registers from consecutive memory locations using an address from an ARM core register as the base address.

Restrictions

The restrictions are:

*   If *size* is present, it must be equal to the size in bits, 32 or 64, of the registers in *list*.
*   For the base address, the SP can be used. In the ARM instruction set, if ! is not specified the PC can be used.
*   *list* must contain at least one register. If it contains doubleword registers, it must not contain more than 16 registers.
*   If using the `Decrement before` addressing mode, the write back flag, *!*, must be appended to the base register specification.

Condition flags

These instructions do not change the flags.

Example

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

3.11.11 VLDR

Loads a single extension register from memory.

Syntax

VLDR{*cond*}{*.F<32|64>*} *<Sd|Dd>*, [*Rn* {, #*imm*}]

VLDR{*cond*}{*.F<32|64>*} *<Sd|Dd>*, *label*

VLDR{*cond*}{*.F<32|64>*} *<Sd|Dd>*, [PC, #*imm*]

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.12 VMLA and VMLS

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

Syntax

```
VMLA{cond}.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>

VMLS{cond}.F<32|64> <Sd|Dd>, <Sn|Dn>, <Sm|Dm>
```

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sn|Dn>, <Sm|Dm>

              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.

3.11.13 VMOV Immediate

Move floating-point Immediate.

Syntax

```
VMOV{cond}.F<32|64> <Sd|Dd>, #imm
```

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   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.

3.11.14 VMOV Register

Copies the contents of one register to another.

Syntax

`VMOV{cond}.F<32|64> <Sd|Dd>, <Sm|Dm>`

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.15 VMOV Scalar to ARM Core register

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

Syntax

VMOV{*cond*} *Rt*, *Dn*[*x*]

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.16 VMOV ARM Core register to single-precision

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

Syntax

VMOV{*cond*} *Sn, Rt*

VMOV{*cond*} *Rt, Sn*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<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.

3.11.17 VMOV two ARM Core registers to two single-precision registers

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

Syntax

VMOV{*cond*} *Sm*, *Sm1*, *Rt*, *Rt2*

VMOV{*cond*} *Rt*, *Rt2*, *Sm*, *Sm1*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.18 VMOV two ARM core registers and a double-precision register

Transfers two words from two ARM core registers to a doubleword register, or from a doubleword register to two ARM core registers.

Syntax

```
VMOV{cond} Dm, Rt, Rt2

VMOV{cond} Rt, Rt2, Dm
```

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.19 VMOV ARM Core register to scalar

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

Syntax

VMOV{*cond*}{*.32*} *Dd*[*x*], *Rt*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.20 VMRS

Move to ARM Core register from floating-point System Register.

Syntax

VMRS{*cond*} *Rt*, FPSCR

VMRS{*cond*} *APSR_nzcv*, FPSCR

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.21 VMSR

Move to floating-point System Register from ARM Core register.

Syntax

VMSR{*cond*} FPSCR, *Rt*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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. See *Floating-point Status Control Register* on page 4-58 for more information.

Restrictions

Rt cannot be PC or SP.

Condition flags

This instruction updates the FPSCR.

3.11.22 VMUL

Floating-point Multiply.

Syntax

VMUL{*cond*}.F<32|64> {<*Sd|Dd*>,} <*Sn|Dn*>, <*Sm|Dm*>

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sn|Dn>, <Sm|Dm>

        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.

3.11.23 VNEG

Floating-point Negate.

Syntax

VNEG{*cond*}.F<32|64> *<Sd|Dd>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sm|Dm>   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.

3.11.24 VNMLA, VNMLS and VNMUL

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

Syntax

VNMLA{*cond*}.F<32|64> *<Sd|Dd>*, *<Sn|Dn>*, *<Sm|Dm>*

VNMLS{*cond*}.F<32|64> *<Sd|Dd>*, *<Sn|Dn>*, *<Sm|Dm>*

VNMUL{*cond*}.F<32|64> {*<Sd|Dd>*,} *<Sn|Dn>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point register.

<Sn|Dn>, <Sm|Dm>

                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.

3.11.25 VPOP

Floating-point extension register Pop.

Syntax

`VPOP{cond}{.size} list`

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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

The list must contain at least one register, and not more than sixteen registers.

Condition flags

These instructions do not change the flags.

3.11.26 VPUSH

Floating-point extension register Push.

Syntax

```
VPUSH{cond}{.size} list
```

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

3.11.27 VSQRT

Floating-point Square Root.

Syntax

VSQRT{*cond*}.F<32|64> *<Sd|Dd>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sm|Dm>   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.

3.11.28 VSTM

Floating-point Store Multiple.

Syntax

VSTM{*mode*}{*cond*}{*.size*} *Rn*{!}, *list*

Where:

mode	Is the addressing mode:

IA   *Increment after*. The consecutive addresses start at the address specified in *Rn*. This is the default and can be omitted.

DB   *Decrement before*. The consecutive addresses end just before the address specified in *Rn*.

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

3.11.29 VSTR

Floating-point Store.

Syntax

VSTR{*cond*}{*.32*} *Sd*, [*Rn*{, #*imm*}]

VSTR{*cond*}{*.64*} *Dd*, [*Rn*{, #*imm*}]

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
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.

3.11.30 VSUB

Floating-point Subtract.

Syntax

VSUB{*cond*}.F<32|64> {<*Sd|Dd*>,} <*Sn|Dn*>, <*Sm|Dm*>

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

<Sd|Dd>   Is the destination floating-point value.

<Sn|Dn>, <Sm|Dm>

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.

3.11.31 VSEL

Provides an alternative to a pair of conditional VMOV instructions according to the condition codes in the GE flags.

Syntax

VSEL{*cond*}.F<32|64> *<Sd|Dd>*, *<Sn|Dn>*, *<Sm|Dm>*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20. 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|Dd>   Is the destination single-precision or double-precision floating-point value.

<Sn|Dn>, <Sm|Dm>

                  Are the operand single-precision or double-precision floating-point values.

Operation

Depending on the result of the condition code, this instruction moves either:

- <Sn|Dn> source register to the destination register.
- <Sm|Dm> source register to the destination register.

The behavior is:

```
EncodingSpecificOperations();
ExecuteFPCheck();

if dp_operation then
   D[d] = if ConditionHolds(cond) then D[n] else D[m];
else
   S[d] = if ConditionHolds(cond) then S[n] else S[m];
```

Restrictions

The VSEL instruction must not occur inside an IT block.

Condition flags

These instructions do not change the flags.

3.11.32 VMAXNM and VMINNM

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

Syntax

`VMAXNM.F<32|64>` *`<Sd|Dd>`*, *`<Sn|Dn>`*, *`<Sm|Dm>`*

`VMINNM.F<32|64>` *`<Sd|Dd>`*, *`<Sn|Dn>`*, *`<Sm|Dm>`*

Where:

`<Sd|Dd>`   Is the destination single-precision or double-precision floating-point value.

`<Sn|Dn>`, `<Sm|Dm>`

Are the operand single-precision or double-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 lowest to the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

3.11.33 VCVTA, VCVTN, VCVTP and VCVTM

Floating-point to integer conversion with directed rounding.

Syntax

VCVT*<rmode>*.S32.F<32|64> *<Sd>*, *<Sm|Dm>*

VCVT*<rmode>*.U32.F<32|64> *<Sd>*, *<Sm|Dm>*

Where:

<Sd|Dd>   Is the destination single-precision or double-precision floating-point value.

<Sn|Dn>, <Sm|Dm>

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.

3.11.34 VRINTR and VRINTX

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

Syntax

VRINT{R,X}{*cond*}.F<32|64> *<Sd|Dd>*, *<Sm|Dm>*

Where:

cond	Is an optional condition code. See *Conditional execution* on page 3-20.
<Sd|Dd>	Is the destination floating-point value.
<Sm|Dm>	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.

3. Write the result to the destination register.

4. For the VRINTZX instruction only. Generate a floating-point exception if the result is not exact.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

3.11.35 VRINTA, VRINTN, VRINTP, VRINTM, and VRINTZ

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

Syntax

VRINT*<rmode>*.F<32|64> *<Sd|Dd>*, *<Sm|Dm>*

Where:

<Sd|Dd>   Is the destination single-precision or double-precision floating-point value.

<Sn|Dn>, <Sm|Dm>

Are the operand single-precision or double-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.   Write the result to the destination register.

Restrictions

These instructions cannot be conditional. These instructions cannot generate an inexact exception even if the result is not exact.

Condition flags

These instructions do not change the flags.

3.12 Miscellaneous instructions

Table 3-16 shows the remaining Cortex-M7 instructions:

Table 3-16 Miscellaneous instructions

Mnemonic	Brief description	See
BKPT	Breakpoint	*BKPT* on page 3-173
CPSID	Change Processor State, Disable Interrupts	*CPS* on page 3-174
CPSIE	Change Processor State, Enable Interrupts	*CPS* on page 3-174
DMB	Data Memory Barrier	*DMB* on page 3-175
DSB	Data Synchronization Barrier	*DSB* on page 3-176
ISB	Instruction Synchronization Barrier	*ISB* on page 3-177
MRS	Move from special register to register	*MRS* on page 3-178
MSR	Move from register to special register	*MSR* on page 3-179
NOP	No Operation	*NOP* on page 3-181
SEV	Send Event	*SEV* on page 3-182
SVC	Supervisor Call	*SVC* on page 3-183
WFE	Wait For Event	*WFE* on page 3-184
WFI	Wait For Interrupt	*WFI* on page 3-185

3.12.1   BKPT

Breakpoint.

Syntax

BKPT *#imm*

Where:

imm   Is an expression evaluating to an integer in the range 0-255 (8-bit value).

Operation

The BKPT instruction causes the processor to enter Debug state. 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.

Examples

```
BKPT #0x3   ; Breakpoint with immediate value set to 0x3 (debugger can
            ; extract the immediate value by locating it using the PC)
```

——— **Note** ———

ARM does not recommend the use of the BKPT instruction with an immediate value set to 0xAB for any purpose other than Semi-hosting.

3.12.2 CPS

Change Processor State.

Syntax

`CPSeffect 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. See *Exception mask registers* on page 2-7 for more information about these registers.

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.

Examples

```
CPSID i  ; Disable interrupts and configurable fault handlers (set PRIMASK)
CPSID f  ; Disable interrupts and all fault handlers (set FAULTMASK)
CPSIE i  ; Enable interrupts and configurable fault handlers (clear PRIMASK)
CPSIE f  ; Enable interrupts and fault handlers (clear FAULTMASK)
```

3.12.3 CPY

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

Syntax

`CPY Rd, Rn`

This is equivalent to:

`MOV Rd, Rn`

3.12.4   DMB

Data Memory Barrier.

Syntax

DMB{*cond*} {*opt*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
DMB  ; Data Memory Barrier
```

3.12.5   DSB

Data Synchronization Barrier.

Syntax

DSB{*cond*} {*opt*}

Where:

cond Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
DSB ; Data Synchronisation Barrier
```

3.12.6   ISB

Instruction Synchronization Barrier.

Syntax

ISB{*cond*} {*opt*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
ISB  ; Instruction Synchronisation Barrier
```

3.12.7 MRS

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

Syntax

MRS{*cond*} *Rd*, *spec_reg*

Where:

cond　　　Is an optional condition code. See *Conditional execution* on page 3-20.

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, or CONTROL.

———— **Note** ————

All the EPSR and IPSR fields are zero when read by the MRS instruction.

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.

See *MSR* on page 3-179.

Restrictions

Rd must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

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

3.12.8 MSR

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

Syntax

MSR{*cond*} *spec_reg*, *Rn*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

Rn   Is the source register.

spec_reg   Can be any of: APSR_nzcvq, APSR_g, APSR_nzcvqg, MSP, PSP, PRIMASK, BASEPRI,
              BASEPRI_MAX, FAULTMASK, or CONTROL.

> ──── **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 Table 2-4 on page 2-5. Privileged software can access all special registers.

In unprivileged software writes to unallocated or execution state bits in the PSR are ignored.

> ──── **Note** ────
>
> When you write to BASEPRI_MAX, the instruction writes to BASEPRI only if either:
>
> *   *Rn* is non-zero and the current BASEPRI value is 0.
> *   *Rn* is non-zero and less than the current BASEPRI value.

See *MRS* on page 3-178.

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*.

Examples

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

3.12.9   NEG

Negate is a pre-UAL synonym for RSB with an immediate value of 0.

Syntax

NEG{*cond*} {*Rd*,} *Rm*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

This is equivalent to:

RSBS{*cond*} {*Rd*,} *Rm*, #0

3.12.10 NOP

No Operation.

Syntax

NOP{*cond*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

```
NOP  ; No operation
```

3.12.11 SEV

Send Event.

Syntax

SEV{*cond*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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, see *Power management* on page 2-30.

Condition flags

This instruction does not change the flags.

Examples

```
SEV ; Send Event
```

3.12.12 SVC

Supervisor Call.

Syntax

SVC{*cond*} #*imm*

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Examples

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

3.12.13 WFE

Wait For Event.

Syntax

WFE{*cond*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

For more information see *Power management* on page 2-30.

Condition flags

This instruction does not change the flags.

Examples

```
WFE  ; Wait for event
```

3.12.14   WFI

Wait for Interrupt.

Syntax

WFI{*cond*}

Where:

cond   Is an optional condition code. See *Conditional execution* on page 3-20.

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.

Condition flags

This instruction does not change the flags.

Examples

```
WFI ; Wait for interrupt
```

Chapter 4
Cortex-M7 Peripherals

This section describes the Cortex-M7 core peripherals. It contains the following sections:

4.1   About the Cortex-M7 peripherals

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

Table 4-1 Core peripheral register regions

Address	Core peripheral	Description
`0xE000E008-0xE000E00F`	System control block	Table 4-12 on page 4-11
`0xE000E010-0xE000E01F`	System timer	Table 4-33 on page 4-33
`0xE000E100-0xE000E4EF`	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
`0xE000ED00-0xE000ED3F`	System control block	Table 4-12 on page 4-11
`0xE000ED78-0xE000ED84`	Processor features[a]	Table 4-39 on page 4-37
`0xE000ED90-0xE000EDB8`	Memory Protection Unit[ab]	Table 4-48 on page 4-44
`0xE000EF00-0xE000EF03`	Nested Vectored Interrupt Controller	Table 4-2 on page 4-3
`0xE000EF30-0xE000EF44`	Floating Point Unit[a]	Table 4-58 on page 4-55
`0xE000EF50-0xE000EF78`	Cache maintenance operations[a]	Table 4-64 on page 4-61
`0xE000EF90-0xE000EFA8`	Access control[a]	Table 4-68 on page 4-66

a. This core peripheral is an optional implementation.

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

In register descriptions:
*   the register *type* is described as follows:

    RW   Read and write.

    RO   Read-only.

    WO   Write-only.
*   the *required privilege* gives the privilege level 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.

——— **Note** ———

Attempting to access a privileged register from unprivileged software results in a BusFault.

4.2 Nested Vectored Interrupt Controller

This section describes the NVIC and the registers it uses. The NVIC supports:

- An implementation-defined number of interrupts in the range 1-240.

- An implementation-defined programmable priority level of 0-255 for each interrupt. A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.

- Level and pulse detection of interrupt signals.

- Dynamic reprioritization of interrupts.

- Grouping of priority values into group priority and subpriority fields.

- Interrupt tail-chaining.

- An external *Non-Maskable Interrupt* (NMI).

- Optional WIC, providing ultra-low power sleep mode support.

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 hardware implementation of the NVIC registers is:

Table 4-2 NVIC register summary

Address[a]	Name[b]	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt Set-enable Registers* on page 4-4
0XE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt Clear-enable Registers* on page 4-5
0XE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt Set-pending Registers* on page 4-5
0XE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt Clear-pending Registers* on page 4-6
0xE000E300-0xE000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt Active Bit Registers* on page 4-7
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt Priority Registers* on page 4-7
0xE000EF00	STIR	WO	Configurable[c]	0x00000000	*Software Trigger Interrupt Register* on page 4-8

a. The address ranges are implementation defined.

b. The range of registers is implementation defined.

c. See the register description for more information.

4.2.1 Accessing the Cortex-M7 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 4-3 CMSIS access NVIC functions
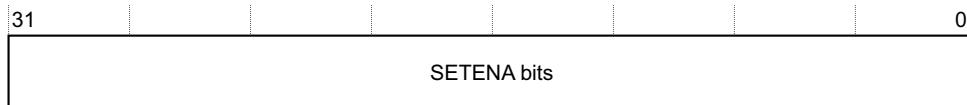
CMSIS function	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)[a]	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)[a]	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)[a]	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)[a]	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)[a]	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)[a]	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)[a]	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

a. The input parameter IRQn is the IRQ number, see Table 2-14 on page 2-20 for more information.

4.2.2 Interrupt Set-enable Registers

The NVIC_ISER0-NVIC_ISER7 registers enable interrupts, and show which interrupts are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

31							0
			SETENA bits				

Table 4-4 ISER bit assignments
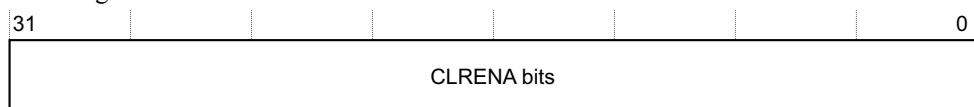
Bits	Name	Function	
[31:0]	SETENA	Interrupt set-enable bits.	
		Write:	
		0	No effect.
		1	Enable interrupt.
		Read:	
		0	Interrupt disabled.
		1	Interrupt enabled.

If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

4.2.3   Interrupt Clear-enable Registers

The NVIC_ICER0-NVIC_ICER7 registers disable interrupts, and show which interrupts are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

31								0
			CLRENA bits					
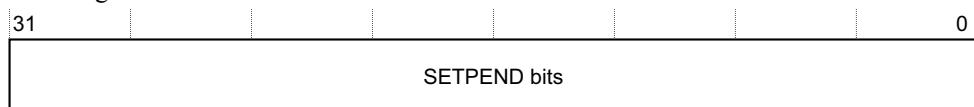
Table 4-5 ICER bit assignments

Bits	Name	Function
[31:0]	CLRENA	Interrupt clear-enable bits.
		Write:
		0 No effect.
		1 Disable interrupt.
		Read:
		0 Interrupt disabled.
		1 Interrupt enabled.

4.2.4   Interrupt Set-pending Registers

The NVIC_ISPR0-NVIC_ISPR7 registers force interrupts into the pending state, and show which interrupts are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

31								0
			SETPEND bits					

Table 4-6 ISPR bit assignments

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits.
		Write:
		0 No effect.
		1 Changes interrupt state to pending.
		Read:
		0 Interrupt is not pending.
		1 Interrupt is pending.

—— **Note** ——
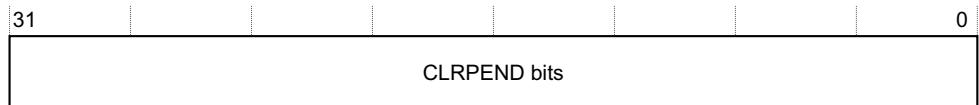
Writing 1 to the ISPR bit corresponding to:

- An interrupt that is pending has no effect.
- A disabled interrupt sets the state of that interrupt to pending.

4.2.5   Interrupt Clear-pending Registers

The NVIC_ICPR0-NCVIC_ICPR7 registers remove the pending state from interrupts, and show which interrupts are pending. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

31							0
			CLRPEND bits				

Table 4-7 ICPR bit assignments

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits.
		Write:
		0 No effect.
		1 Removes pending state an interrupt.
		Read:
		0 Interrupt is not pending.
		1 Interrupt is pending.
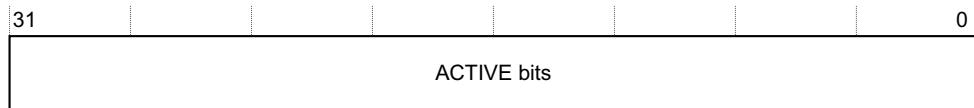
——— **Note** ———

Writing 1 to an ICPR bit does not affect the active state of the corresponding interrupt.

4.2.6 Interrupt Active Bit Registers

The NVIC_IABR0-NVIC_IABR7 registers indicate which interrupts are active. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

| 31							0
			ACTIVE bits				
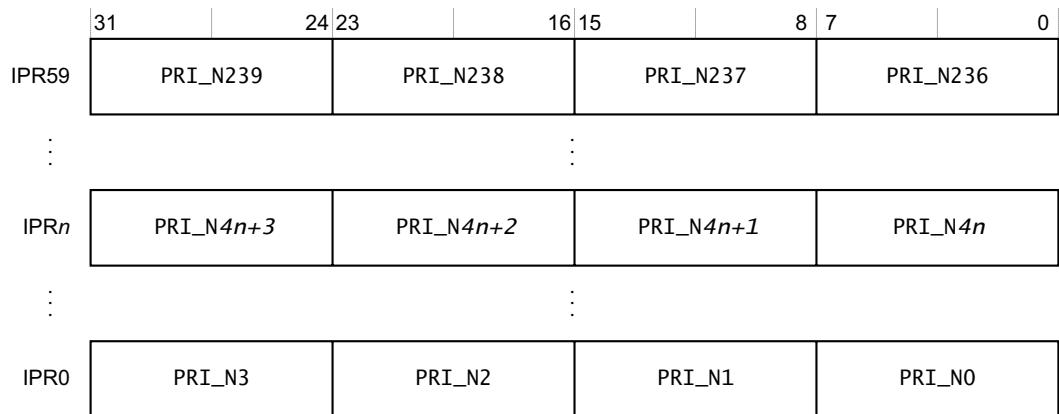
Table 4-8 IABR bit assignments

Bits	Name	Function	
[31:0]	ACTIVE	Interrupt active flags:	
		0	Interrupt not active.
		1	Interrupt active.

A bit reads as one if the status of the corresponding interrupt is active or active and pending.

4.2.7 Interrupt Priority Registers

The NVIC_IPR0-NVIC_IPR59 registers provide a priority field for each interrupt. These registers are byte-accessible. See the register summary in Table 4-2 on page 4-3 for their attributes. Each register holds four priority fields as shown:

	31 24	23 16	15 8	7 0
IPR59	PRI_N239	PRI_N238	PRI_N237	PRI_N236
⋮				
IPRn	PRI_N4n+3	PRI_N4n+2	PRI_N4n+1	PRI_N4n
⋮				
IPR0	PRI_N3	PRI_N2	PRI_N1	PRI_N0

Table 4-9 IPR bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. If enabled, the processor can implement only bits[7:n] of each field, bits[n-x:0] read as zero and ignore writes. The values of n and x are implementation defined.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

See *Accessing the Cortex-M7 NVIC registers using CMSIS* on page 4-4 for more information about the access to the interrupt priority array, that provides the software view of the interrupt priorities.

Find the IPR number and byte offset for interrupt *m* as follows:

- the corresponding IPR number, see Table 4-8 on page 4-7 *n* is given by *n = m* 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].

4.2.8 Software Trigger Interrupt Register
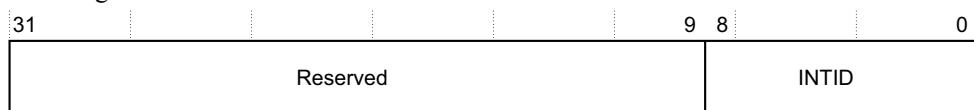
Write to the STIR to generate an interrupt from software. See the register summary in Table 4-2 on page 4-3 for the STIR attributes.

When the USERSETMPEND bit in the SCR is set to 1, unprivileged software can access the STIR, see *System Control Register* on page 4-20.

Note

Only privileged software can enable unprivileged access to the STIR.

The bit assignments are:

31	9	8	0
Reserved		INTID	

Table 4-10 STIR bit assignments

Bits	Field	Function
[31:9]	-	Reserved.
[8:0]	INTID	Interrupt ID of the interrupt to trigger, in the range 0-239. For example, a value of `0x03` specifies interrupt IRQ3.

4.2.9 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 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, see *Hardware and software control of interrupts* on page 4-9. 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.

See the documentation supplied by your device vendor for details of which interrupts are level-based and which are pulsed.

Hardware and software control of interrupts

The Cortex-M7 processor latches all interrupts. A peripheral interrupt becomes pending for one of the following reasons:

- The NVIC detects that the interrupt signal is HIGH and the interrupt is not active.

- The NVIC detects a rising edge on the interrupt signal.

- Software writes to the corresponding interrupt set-pending register bit, see *Interrupt Set-pending Registers* on page 4-5, or to the STIR to make an interrupt pending, see *Software Trigger Interrupt Register* on page 4-8.

A pending interrupt remains pending until one of the following:

- 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 re-enter 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 re-enter 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, the state of the interrupt changes to:
  - Inactive, if the state was pending.
  - Active, if the state was active and pending.

4.2.10 NVIC design hints and tips

Ensure software uses correctly aligned register accesses. The processor does not support unaligned accesses to NVIC registers. See the individual register descriptions for the supported access sizes.

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 the vector table entries of the new vector table are set up for fault handlers, NMI, and all enabled exception-like interrupts. For more information see *Vector Table Offset Register* on page 4-17.

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 a number of functions for NVIC control, including:

Table 4-11 CMSIS functions for NVIC control

CMSIS interrupt control function	Description
`void NVIC_SetPriorityGrouping(uint32_t priority_grouping)`	Set the priority grouping
`void NVIC_EnableIRQ(IRQn_t IRQn)`	Enable IRQn
`void NVIC_DisableIRQ(IRQn_t IRQn)`	Disable IRQn
`uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)`	Return true (IRQ-Number) if IRQn is pending
`void NVIC_SetPendingIRQ (IRQn_t IRQn)`	Set IRQn pending
`void NVIC_ClearPendingIRQ (IRQn_t IRQn)`	Clear IRQn pending status
`uint32_t NVIC_GetActive (IRQn_t IRQn)`	Return the IRQ number of the active interrupt
`void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)`	Set priority for IRQn
`uint32_t NVIC_GetPriority (IRQn_t IRQn)`	Read priority of IRQn

The input parameter IRQn is the IRQ number, see Table 2-14 on page 2-20. For more information about these functions see the CMSIS documentation.

4.3 System control block

The *System Control Block* (SCB) provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions. The system control block registers are:

Table 4-12 Summary of the system control block registers

Address	Name	Type	Required privilege	Reset value	Description
0xE000E008	ACTLR	RW	Privileged	0x00000000	*Auxiliary Control Register*
0xE000ED00	CPUID	RO	Privileged	0x410FC270[a]	*CPUID Base Register* on page 4-13
0xE000ED04	ICSR	RW[b]	Privileged	0x00000000	*Interrupt Control and State Register* on page 4-14
0xE000ED08	VTOR	RW	Privileged	Unknown	*Vector Table Offset Register* on page 4-17
0xE000ED0C	AIRCR	RW[b]	Privileged	0xFA050000	*Application Interrupt and Reset Control Register* on page 4-17
0xE000ED10	SCR	RW	Privileged	0x00000000	*System Control Register* on page 4-20
0xE000ED14	CCR	RW	Privileged	0x00000200[a]	*Configuration and Control Register* on page 4-20
0xE000ED18	SHPR1	RW	Privileged	0x00000000	*System Handler Priority Register 1* on page 4-23
0xE000ED1C	SHPR2	RW	Privileged	0x00000000	*System Handler Priority Register 2* on page 4-23
0xE000ED20	SHPR3	RW	Privileged	0x00000000	*System Handler Priority Register 3* on page 4-23
0xE000ED24	SHCRS	RW	Privileged	0x00000000	*System Handler Control and State Register* on page 4-24
0xE000ED28	CFSR	RW	Privileged	0x00000000	*Configurable Fault Status Register* on page 4-25
0xE000ED28	MMSR[c]	RW	Privileged	0x00	*MemManage Fault Status Register* on page 4-26
0xE000ED29	BFSR[c]	RW	Privileged	0x00	*BusFault Status Register* on page 4-27
0xE000ED2A	UFSR[c]	RW	Privileged	0x0000	*UsageFault Status Register* on page 4-29
0xE000ED2C	HFSR	RW	Privileged	0x00000000	*HardFault Status Register* on page 4-31
0xE000ED34	MMAR	RW	Privileged	Unknown	*MemManage Fault Address Register* on page 4-31
0xE000ED38	BFAR	RW	Privileged	Unknown	*BusFault Address Register* on page 4-32
0xE000ED3C	AFSR	RAZ/WI	Privileged	-	Auxiliary Fault Status Register not implemented

a. This reset value is implementation defined.
b. See the register description for more information.
c. A subregister of the CFSR.
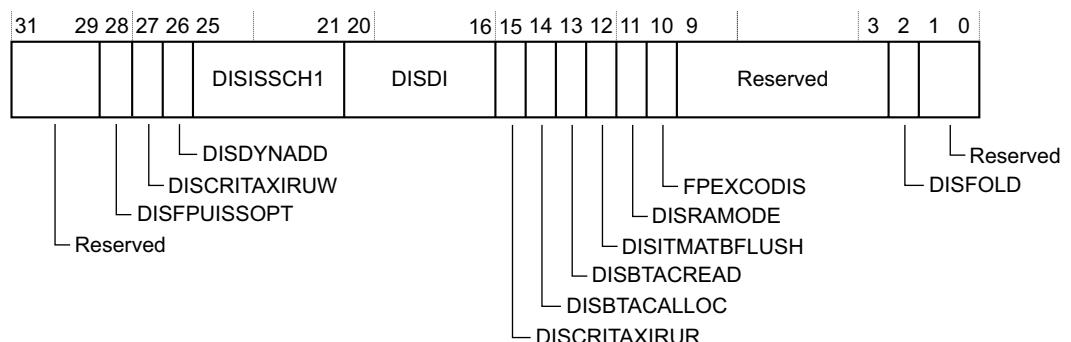
4.3.1 Auxiliary Control Register

The ACTLR provides disable bits for the following processor functions:
- FPU exception outputs.
- Dual-issue functionality.
- Flushing of the trace output from the ITM and DWT.
- Dynamic read allocate mode.

By default this register is set to provide optimum performance from the Cortex-M7 processor, and does not normally require modification.

See the register summary in Table 4-12 on page 4-11 for the ACTLR attributes. The bit assignments are:

Table 4-13 ACTLR bit assignments

Bits	Name	Function	
[31:29]	-	Reserved.	
[28]	DISFPUISSOPT	**0**	Normal operation.
[27]	DISCRITAXIRUW	Disable critical AXI read-under-write:	
		0	Normal operation. This is backwards compatible with r0.
		1	AXI reads to DEV/SO memory. Exclusive reads to shared memory are not initiated on the AXIM AR channel until all outstanding stores on AXI are complete.
[26]	DISDYNADD	Disables dynamic allocation of ADD and SUB instructions:	
		0	Normal operation. Some ADD and SUB instructions are resolved in EX1.
		1	All ADD and SUB instructions are resolved in EX2.
[25:21]	DISISSCH1	**0**	Normal operation.
		1	Instruction type might not be issued in channel 1.
		[25]	VFP.
		[24]	Integer MAC and MUL.
		[23]	Loads to PC.
		[22]	Indirect branches, but not loads to PC.
		[21]	Direct branches.
[20:16]	DISDI	**0**	Normal operation.
		1	Nothing can be dual-issued when this instruction type is in channel 0.
		[20]	VFP.
		[19]	Integer MAC and MUL.
		[18]	Loads to PC.
		[17]	Indirect branches, but not loads to PC.
		[16]	Direct branches.
[15]	DISCRITAXIRUR	Disables critical AXI Read-Under-Read.	
		0	Normal operation.
		1	An AXI read to Strongly-ordered or device memory, or an LDREX to shared memory, is not put on AXI if there are any outstanding reads on AXI. Transactions on AXI cannot be interrupted. This bit might reduce the time that these transactions are in progress and might improve worst case interrupt latency. Performance is decreased when this bit is set.

Table 4-13 ACTLR bit assignments (continued)

Bits	Name	Function	
[14]	DISBTACALLOC	**0**	Normal operation.
		1	No new entries are allocated in *Branch Target Address Cache* (BTAC), but existing entries can be updated.
[13]	DISBTACREAD	**0**	Normal operation.
		1	BTAC is not used and only static branch prediction can occur.
[12]	DISITMATBFLUSH	Disables ITM and DWT ATB flush:	
		1	ITM and DWT ATB flush disabled. **AFVALID** is ignored and **AFREADY** is held HIGH.
		——— **Note** ———	
		This bit is always 1 and therefore RO/WI.	
[11]	DISRAMODE	Disables dynamic read allocate mode for Write-Back Write-Allocate memory regions:	
		0	Normal operation.
		1	Dynamic disabled.
		See *Dynamic read allocate mode* on page 2-18.	
[10]	FPEXCODIS	Disables FPU exception outputs:	
		0	Normal operation.
		1	FPU exception outputs are disabled.
[9:3]	-	Reserved.	
[2]	DISFOLD	**0**	Normal operation.
[1:0]	-	Reserved.	

4.3.2 CPUID Base Register

The CPUID register contains the processor part number, version, and implementation information. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

31	24	23	20	19	16	15	4	3	0
Implementer		Variant		Constant		PartNo		Revision	
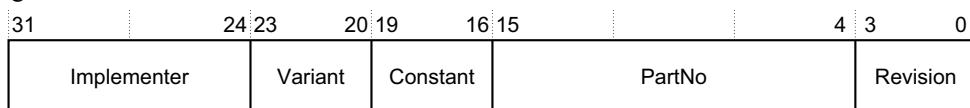
Table 4-14 CPUID bit assignments

Bits	Name	Function	
[31:24]	Implementer	Implementer code:	
		`0x41`	ARM
[23:20]	Variant	Variant number, the r value in the r*m*p*n* product revision identifier:	
		`0x0`	Revision 0
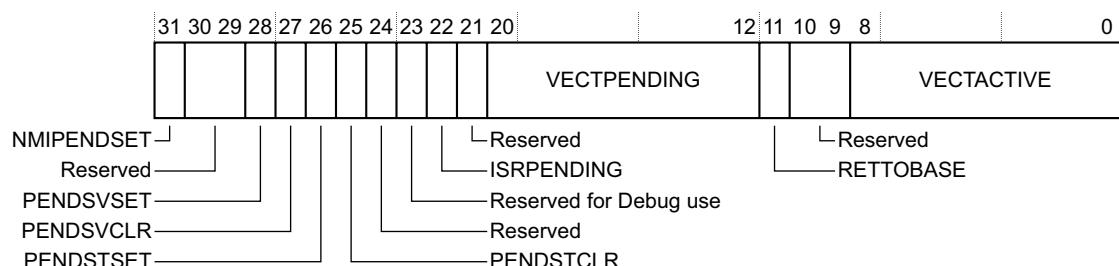
Bits	Name	Function
[19:16]	Constant	Reads as `0xF`
[15:4]	PartNo	Part number of the processor:
		`0xC27` Cortex-M7
[3:0]	Revision	Revision number, the p value in the r*n*p*n* product revision identifier:
		`0x0` Patch 0

4.3.3 Interrupt Control and State Register

The ICSR:

- provides:
  — A set-pending bit for the *Non-Maskable Interrupt* (NMI) exception.
  — Set-pending and clear-pending bits for the PendSV and SysTick exceptions.
- indicates:
  — The exception number of the exception being processed.
  — Whether there are preempted active exceptions.
  — The exception number of the highest priority pending exception.
  — Whether any interrupts are pending.

See the register summary in Table 4-12 on page 4-11, and the Type descriptions in Table 4-15 on page 4-15, for the ICSR attributes. The bit assignments are:

Table 4-15 ICSR bit assignments

Bits	Name	Type	Function
[31]	NMIPENDSET	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. Because NMI is the highest-priority exception, normally the processor enters the NMI exception handler as soon as it registers a write of 1 to this bit, and entering the handler clears this bit to 0. 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.
[30:29]	-	-	Reserved.
[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.
[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]	-	-	Reserved.
[23]	Reserved for Debug use	RO	This bit is reserved for Debug use and reads-as-zero when the processor is not in Debug.
[22]	ISRPENDING	RO	Interrupt pending flag, excluding NMI and Faults: **0** Interrupt not pending. **1** Interrupt pending.
[21]	-	-	Reserved.

Table 4-15 ICSR bit assignments (continued)

Bits	Name	Type	Function
[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 indicated by this field includes the effect of the BASEPRI and FAULTMASK registers, but not any effect of the PRIMASK register.
[11]	RETTOBASE	RO	Indicates whether there are preempted active exceptions:
			0 There are preempted active exceptions to execute.
			1 There are no active exceptions, or the currently-executing exception is the only active exception.
[10:9]	-	-	Reserved.
[8:0]	VECTACTIVE[a]	RO	Contains the active exception number:
			0 Thread mode.
			1 The exception number[a] 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 Table 2-5 on page 2-6.

---
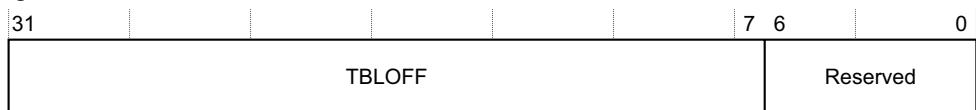
a. This is the same value as IPSR bits[8:0], see *Interrupt Program Status Register* on page 2-6.

When you write to the ICSR, the effect is UNPREDICTABLE if you:

• Write 1 to the PENDSVSET bit and write 1 to the PENDSVCLR bit.
• Write 1 to the PENDSTSET bit and write 1 to the PENDSTCLR bit.

4.3.4 Vector Table Offset Register

The VTOR indicates the offset of the vector table base address from memory address `0x00000000`. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

31					7	6	0
		TBLOFF				Reserved	

Table 4-16 VTOR bit assignments

Bits	Name	Function
[31:7]	TBLOFF	Vector table base offset field. It contains bits[29:7] of the offset of the table base from the bottom of the memory map. The lower bit in the TBLOFF field is implementation defined.
[6:0]	-	Reserved.

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

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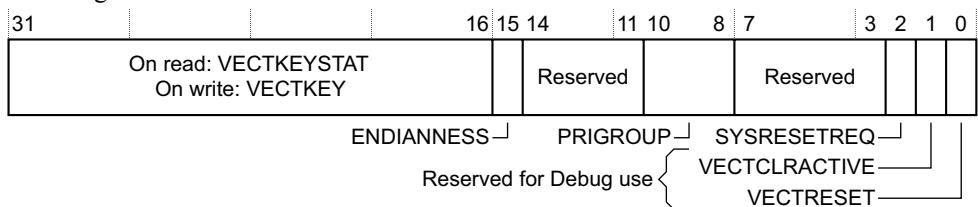
Table alignment requirements mean that bits[6:0] of the table offset are always zero.

4.3.5 Application Interrupt and Reset Control Register

The AIRCR provides priority grouping control for the exception model, endian status for data accesses, and reset control of the system. See the register summary in Table 4-12 on page 4-11 and Table 4-17 on page 4-18 for its attributes.

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

The bit assignments are:

Table 4-17 AIRCR bit assignments

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 setting is implementation defined: **0** Little-endian. **1** Big-endian.
[14:11]	-	-	Reserved.
[10:8]	PRIGROUP	RW	Interrupt priority grouping field. This field determines the split of group priority from subpriority, see *Binary point* on page 4-19.
[7:3]	-	-	Reserved.
[2]	SYSRESETREQ	WO	System reset request bit setting is implementation defined: **0** No system reset request. **1** Asserts a signal to the outer system that requests a reset. This is intended to force a large system reset of all major components except for debug. This bit reads as 0. See your vendor documentation for more information about the use of this signal in your implementation.
[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]	VECTRESET	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.

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. Table 4-18 shows how the PRIGROUP value controls this split. Implementations having PRI_*n* fields of less than 8 bits treat the least-significant bits as zero.

Table 4-18 Priority grouping

	Interrupt priority level value, PRI_*N*[7:0]			Number of	
PRIGROUP	**Binary point[a]**	**Group priority bits**	**Subpriority bits**	**Group priorities[b]**	**Subpriorities[b]**
0b000[c]	bxxxxxxx.y	[7:1]	[0]	128	2
0b001[c]	bxxxxxx.yy	[7:2]	[1:0]	64	4
0b010[c]	bxxxxx.yyy	[7:3]	[2:0]	32	8
0b011[c]	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

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

b. Implementation defined.
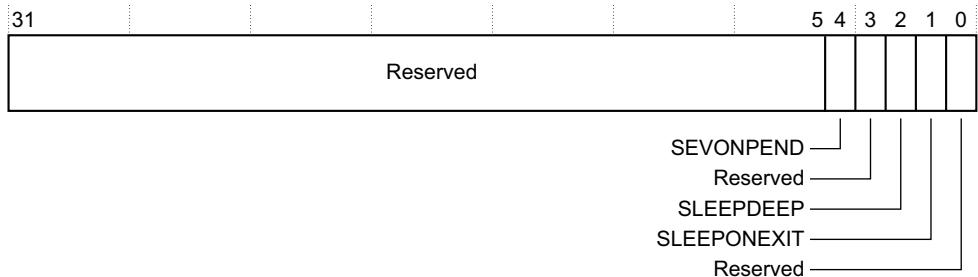
c. Optional.

---- **Note** ----

Determining preemption of an exception uses only the group priority field, see *Interrupt priority grouping* on page 2-23.

4.3.6 System Control Register

The SCR controls features of entry to and exit from low power state. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

Table 4-19 SCR bit assignments

Bits	Name	Function
[31:5]	-	Reserved.
[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]	-	Reserved.
[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.
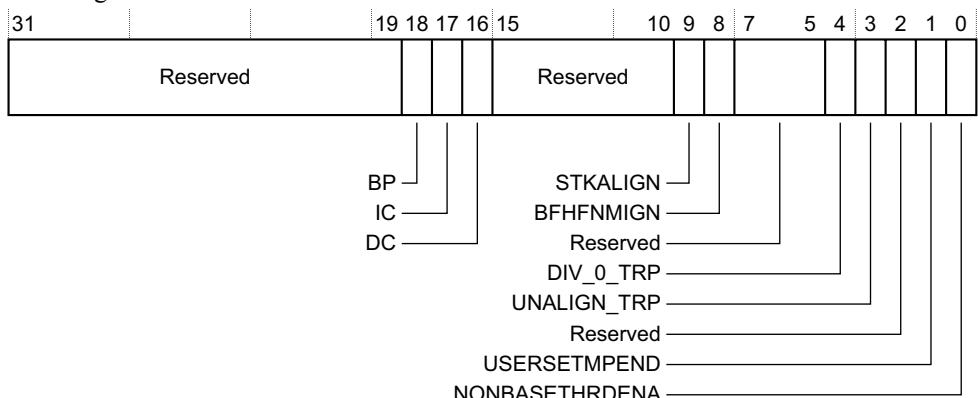
4.3.7 Configuration and Control Register

The CCR controls entry to Thread mode and enables:

- The handlers for NMI, hard fault and faults escalated by FAULTMASK to ignore BusFaults.

- Trapping of divide by zero and unaligned accesses.

- Access to the STIR by unprivileged software, see *Software Trigger Interrupt Register* on page 4-8.

- Optional instruction and data cache enable control.

See the register summary in Table 4-12 on page 4-11 for the CCR attributes.

The bit assignments are:

Table 4-20 CCR bit assignments

Bits	Name	Type	Function
[31:19]	-	-	Reserved.
[18]	BP	RO	Always reads-as-one. It indicates branch prediction is enabled.
[17]	IC	RW	Enables L1 instruction cache. This bit is optional: **0** L1 instruction cache disabled. **1** L1 instruction cache enabled.
[16]	DC	RW	Enables L1data cache. This bit is optional: **0** L1 data cache disabled. **1** L1 data cache enabled.
[15:10]	-	-	Reserved.
[9]	STKALIGN	RO	Always reads-as-one. It indicates stack alignment on exception entry is 8-byte aligned. On exception entry, the processor uses bit[9] of the stacked PSR to indicate the stack alignment. On return from the exception it uses this stacked bit to restore the correct stack alignment.
[8]	BFHFNMIGN	RW	Enables handlers with priority -1 or -2 to ignore data BusFaults caused by load and store instructions. This applies to the hard fault, NMI, and FAULTMASK escalated handlers: **0** Data bus faults caused by load and store instructions cause a lock-up. **1** Handlers running at priority -1 and -2 ignore data bus faults caused by load and store instructions. Set this bit to 1 only when the handler and its data are in absolutely safe memory. The normal use of this bit is to probe system devices and bridges to detect control path problems and fix them.
[7:5]	-	-	Reserved.
[4]	DIV_0_TRP	RW	Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0: **0** Do not trap divide by 0. **1** Trap divide by 0. When this bit is set to 0, a divide by zero returns a quotient of 0.

Table 4-20 CCR bit assignments (continued)

Bits	Name	Type	Function
[3]	UNALIGN_TRP	RW	Enables unaligned access traps: **0** Do not trap unaligned halfword and word accesses. **1** Trap unaligned halfword and word accesses. If this bit is set to 1, an unaligned access generates a UsageFault. Unaligned `LDM`, `STM`, `LDRD`, and `STRD` instructions always fault irrespective of whether UNALIGN_TRP is set to 1.
[2]	-	-	Reserved.
[1]	USERSETMPEND	RW	Enables unprivileged software access to the STIR, see *Software Trigger Interrupt Register* on page 4-8: **0** Disable. **1** Enable.
[0]	NONBASETHRDENA	RW	Indicates how the processor enters Thread mode: **0** Processor can enter Thread mode only when no exception is active. **1** Processor can enter Thread mode from any level under the control of an EXC_RETURN value, see *Exception return* on page 2-26.

4.3.8 System Handler Priority Registers

The SHPR1-SHPR3 registers set the priority level of the exception handlers. The maximum range of configurable priority values is 0-255. The actual range of priority values available is implementation defined.

SHPR1-SHPR3 are byte accessible. See the register summary in Table 4-12 on page 4-11 for their attributes.

The system fault handlers, priority fields and registers are identified in Table 4-21.

Table 4-21 System fault handler priority fields

Handler	Field	Register description
MemManage	PRI_4	*System Handler Priority Register 1* on page 4-23
BusFault	PRI_5	
UsageFault	PRI_6	
SVCall	PRI_11	*System Handler Priority Register 2* on page 4-23
PendSV	PRI_14	*System Handler Priority Register 2* on page 4-23
SysTick	PRI_15	

Each PRI_n field can be up to 8 bits wide, but the processor implements only bits[7:M] of each field, where M is implementation defined. Bits[M-1:0] read as zero and ignore writes.

System Handler Priority Register 1

The bit assignments are:

31	24	23	16	15	8	7	0
Reserved		PRI_6		PRI_5		PRI_4	

Table 4-22 SHPR1 register bit assignments

Bits	Name	Function
[31:24]	PRI_7	Reserved
[23:16]	PRI_6	Priority of system handler 6, UsageFault
[15:8]	PRI_5	Priority of system handler 5, BusFault
[7:0]	PRI_4	Priority of system handler 4, MemManage

System Handler Priority Register 2

The bit assignments are:

31	24	23	0
PRI_11		Reserved	

Table 4-23 SHPR2 register bit assignments

Bits	Name	Function
[31:24]	PRI_11	Priority of system handler 11, SVCall
[23:0]	-	Reserved

System Handler Priority Register 3

The bit assignments are:

31	24	23	16	15	0
PRI_15		PRI_14		Reserved	

Table 4-24 SHPR3 register bit assignments
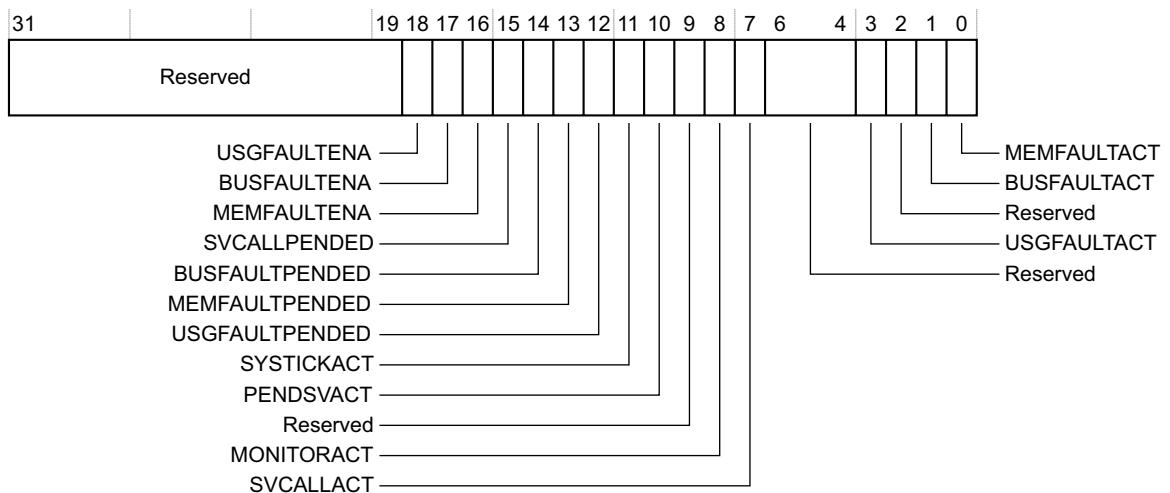
Bits	Name	Function
[31:24]	PRI_15	Priority of system handler 15, SysTick exception
[23:16]	PRI_14	Priority of system handler 14, PendSV
[15:0]	-	Reserved

4.3.9 System Handler Control and State Register

The SHCSR enables the system handlers, and indicates:

- The pending status of the BusFault, MemManage fault, and SVC exceptions.
- The active status of the system handlers.

See the register summary in for the SHCSR attributes. The bit assignments are:

Table 4-25 SHCSR bit assignments

Bits	Name	Function
[31:19]	-	Reserved
[18]	USGFAULTENA	UsageFault enable bit, set to 1 to enable[a]
[17]	BUSFAULTENA	BusFault enable bit, set to 1 to enable[a]
[16]	MEMFAULTENA	MemManage enable bit, set to 1 to enable[a]
[15]	SVCALLPENDED	SVCall pending bit, reads as 1 if exception is pending[b]
[14]	BUSFAULTPENDED	BusFault exception pending bit, reads as 1 if exception is pending[b]
[13]	MEMFAULTPENDED	MemManage exception pending bit, reads as 1 if exception is pending[b]
[12]	USGFAULTPENDED	UsageFault exception pending bit, reads as 1 if exception is pending[b]
[11]	SYSTICKACT	SysTick exception active bit, reads as 1 if exception is active[c]
[10]	PENDSVACT	PendSV exception active bit, reads as 1 if exception is active
[9]	-	Reserved
[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
[6:4]	-	Reserved
[3]	USGFAULTACT	UsageFault exception active bit, reads as 1 if exception is active

Table 4-25 SHCSR bit assignments (continued)

Bits	Name	Function
[2]	-	Reserved
[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

a. Enable bits, set to 1 to enable the exception, or set to 0 to disable the exception.

b. 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.

c. 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.

If you disable a system handler and the corresponding fault occurs, the processor treats the fault as a hard fault.
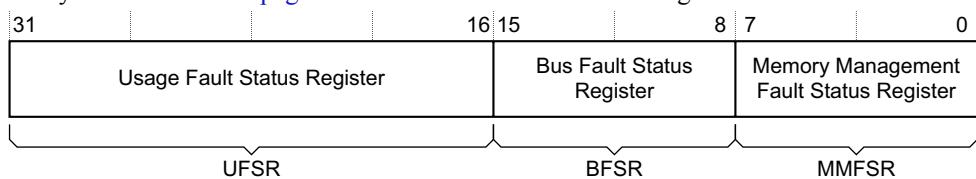
You can write to this register to change the pending or active status of system exceptions. An OS kernel can write to the active bits to perform a context switch that changes the current exception type.

──── **Caution** ────

*   Software that changes the value of an active bit in this register without correct adjustment to the stacked content can cause the processor to generate a fault exception. Ensure software that writes to this register retains and subsequently restores the current active status.

*   After you have enabled the system handlers, if you have to change the value of a bit in this register you must use a read-modify-write procedure to ensure that you change only the required bit.

4.3.10 Configurable Fault Status Register

The CFSR indicates the cause of a MemManage fault, BusFault, or UsageFault. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

31	16	15	8	7	0
Usage Fault Status Register		Bus Fault Status Register		Memory Management Fault Status Register	
UFSR		BFSR		MMFSR	

The following subsections describe the subregisters that make up the CFSR:
*   *MemManage Fault Status Register* on page 4-26.
*   *BusFault Status Register* on page 4-27.
*   *UsageFault Status Register* on page 4-29.

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`.

MemManage Fault Status Register

The flags in the MMFSR indicate the cause of memory access faults. The bit assignments are:
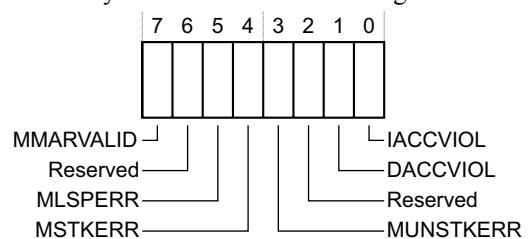
Table 4-26 MMFSR bit assignments

| Bits	Name	Function
[7]	MMARVALID	*MemManage Fault Address Register* (MMFAR) valid flag:
		0 Value in MMAR is not a valid fault address.
		1 MMAR 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 MMAR value has been overwritten.
[6]	-	Reserved.
[5]	MLSPERR[a]	**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 MMAR.
[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 MMAR.
[2]	-	Reserved.
[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 MMAR 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 MMAR.
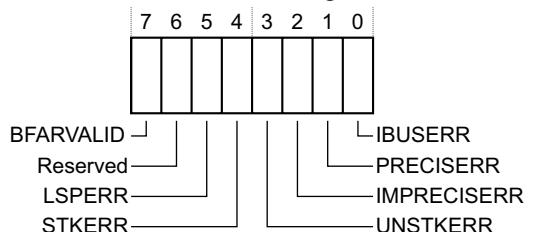
    a.   This field is optional and might not be available in your implementation.

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.

BusFault Status Register

The flags in the BFSR indicate the cause of a bus access fault. The bit assignments are:

```
          7 6 5 4 3 2 1 0
         ┌─┬─┬─┬─┬─┬─┬─┬─┐
         │ │ │ │ │ │ │ │ │
         └─┴─┴─┴─┴─┴─┴─┴─┘
BFARVALID ─┘ │ │ │  │ │ └─ IBUSERR
 Reserved ───┘ │ │  │ └─── PRECISERR
   LSPERR ─────┘ │  └───── IMPRECISERR
   STKERR ───────┘  └───── UNSTKERR
```

Table 4-27 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.	
[5]	LSPERR[a]	**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.	

Table 4-27 BFSR bit assignments (continued)

| Bits	Name	Function
[2]	IMPRECISERR	Imprecise data bus error:
		0 No imprecise data bus error.
		1 A data bus error has occurred, but the return address in the stack frame is not related to the instruction that caused the error.
		When the processor sets this bit to 1, it does not write a fault address to the BFAR.
		This is an asynchronous fault. Therefore, if it is detected when the priority of the current process is higher than the BusFault priority, the BusFault becomes pending and becomes active only when the processor returns from all higher priority processes. If a precise fault occurs before the processor enters the handler for the imprecise BusFault, the handler detects both IMPRECISERR set to 1 and one of the precise fault status bits set to 1.
[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.
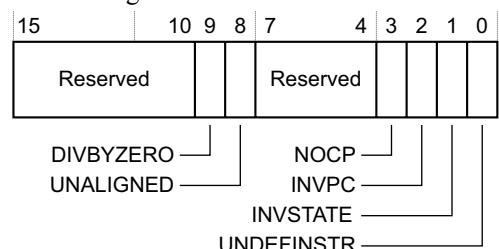
a. This field is optional and might not be available in your implementation.

─────── **Note** ───────

The BFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

UsageFault Status Register

The UFSR indicates the cause of a UsageFault. The bit assignments are:

Table 4-28 UFSR bit assignments

Bits	Name	Function
[15:10]	-	Reserved.
[9]	DIVBYZERO	Divide by zero UsageFault:
		0 No divide by zero fault, or divide by zero trapping not enabled.
		1 The processor has executed an SDIV or UDIV instruction with a divisor of 0.
		When the processor sets this bit to 1, the PC value stacked for the exception return points to the instruction that performed the divide by zero.
		Enable trapping of divide by zero by setting the DIV_0_TRP bit in the CCR to 1, see *Configuration and Control Register* on page 4-20.
[8]	UNALIGNED	Unaligned access UsageFault:
		0 No unaligned access fault, or unaligned access trapping not enabled.
		1 The processor has made an unaligned memory access.
		Enable trapping of unaligned accesses by setting the UNALIGN_TRP bit in the CCR to 1, see *Configuration and Control Register* on page 4-20.
		Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of the setting of UNALIGN_TRP.
[7:4]	-	Reserved.
[3]	NOCP	No coprocessor UsageFault:
		0 No UsageFault caused by attempting to access a coprocessor.
		1 The processor has attempted to access a coprocessor.
		The processor does not support coprocessor instructions.

Table 4-28 UFSR bit assignments (continued)

Bits	Name	Function
[2]	INVPC	Invalid PC load UsageFault, caused by an invalid PC load by EXC_RETURN:
		0 No invalid PC load UsageFault.
		1 The processor has attempted an illegal load of EXC_RETURN to the PC, as a result of an invalid context, or an invalid EXC_RETURN value.
		When this bit is set to 1, the PC value stacked for the exception return points to the instruction that tried to perform the illegal load of the PC.
[1]	INVSTATE	Invalid state UsageFault:
		0 No invalid state UsageFault.
		1 The processor has attempted to execute an instruction that makes illegal use of the EPSR.
		When this bit is set to 1, the PC value stacked for the exception return points to the instruction that attempted the illegal use of the EPSR.
		This bit is not set to 1 if an undefined instruction uses the EPSR.
[0]	UNDEFINSTR	Undefined instruction UsageFault:
		0 No undefined instruction UsageFault.
		1 The processor has attempted to execute an undefined instruction.
		When this bit is set to 1, the PC value stacked for the exception return points to the undefined instruction.
		An undefined instruction is an instruction that the processor cannot decode.
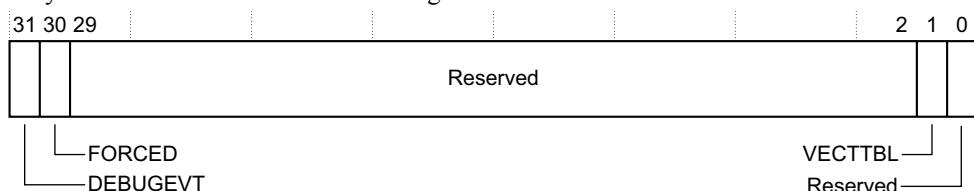
——— **Note** ———

The UFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

4.3.11 HardFault Status Register

The HFSR gives information about events that activate the HardFault handler. See the register summary in Table 4-12 on page 4-11 for its attributes.

This 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. The bit assignments are:

Table 4-29 HFSR bit assignments

Bits	Name	Function
[31]	DEBUGEVT	Reserved for Debug use. When writing to the register you must write 1 to this bit, otherwise behavior is UNPREDICTABLE.
[30]	FORCED	Indicates a forced hard fault, 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.
[1]	VECTTBL	Indicates a BusFault on a vector table read during exception processing: **0** No BusFault on vector table read. **1** BusFault on vector table read. This error is always handled by the hard fault handler. When this bit is set to 1, the PC value stacked for the exception return points to the instruction that was preempted by the exception.
[0]	-	Reserved.

— **Note** —

The HFSR bits are sticky. This means as one or more fault occurs, the associated bits are set to 1. A bit that is set to 1 is cleared to 0 only by writing 1 to that bit, or by a reset.

4.3.12 MemManage Fault Address Register

The MMFAR contains the address of the location that generated a MemManage fault. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

Table 4-30 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. See *MemManage Fault Status Register* on page 4-26.

4.3.13 BusFault Address Register

The BFAR contains the address of the location that generated a BusFault. See the register summary in Table 4-12 on page 4-11 for its attributes. The bit assignments are:

Table 4-31 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. See *BusFault Status Register* on page 4-27.

4.3.14 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.

The processor does not support unaligned accesses to system control block registers.

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 preempts 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 4-32 CMSIS function for system control

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

4.4 System timer, SysTick

The processor has a 24-bit system timer, SysTick, that counts down from the reload value to zero, reloads, that is wraps to, the value in the SYST_RVR register on the next clock edge, then counts down on subsequent clocks.

───── **Note** ─────

When the processor is halted for debugging the counter does not decrement.

The system timer registers are summarized in Table 4-33.

Table 4-33 System timer registers summary
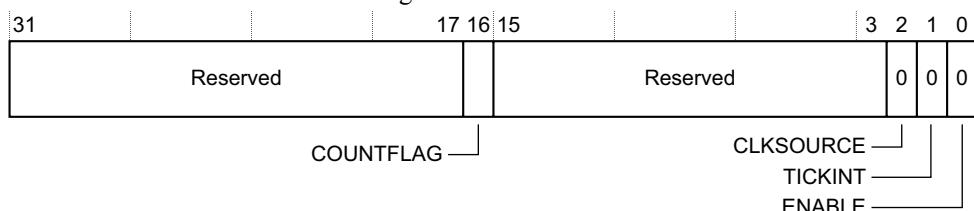
Address	Name	Type	Required privilege	Reset value[a]	Description
0xE000E010	SYST_CSR	RW	Privileged	0x00000004	*SysTick Control and Status Register*
0xE000E014	SYST_RVR	RW	Privileged	UNKNOWN	*SysTick Reload Value Register* on page 4-34
0xE000E018	SYST_CVR	RW	Privileged	UNKNOWN	*SysTick Current Value Register* on page 4-35
0xE000E01C	SYST_CALIB[b]	RO	Privileged	0xC0000000	*SysTick Calibration Value Register* on page 4-35

a. The reset value is implementation defined.

b. The SysTick Calibration Value Register is optional and might not be present in your implementation.

4.4.1 SysTick Control and Status Register

The SysTick SYST_CSR register enables the SysTick features. See the register summary in Table 4-33 for its attributes. The bit assignments are:

Table 4-34 SysTick SYST_CSR bit assignments

Bits	Name	Function
[31:17]	-	Reserved.
[16]	COUNTFLAG	Returns 1 if timer counted to 0 since last time this was read.
[15:3]	-	Reserved.

Bits	Name	Function
[2]	CLKSOURCE	Indicates the clock source:
		0 External 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.
		Software can use COUNTFLAG to determine if SysTick has ever counted to zero.
[0]	ENABLE	Enables the counter:
		0 Counter disabled.
		1 Counter enabled.

When ENABLE is set to 1, the counter loads the RELOAD value from the SYST_RVR register and then counts down. On reaching 0, it sets the COUNTFLAG to 1 and optionally asserts the SysTick depending on the value of TICKINT. It then loads the RELOAD value again, and begins counting.

4.4.2 SysTick Reload Value Register

The SYST_RVR register specifies the start value to load into the SYST_CVR register. See the register summary in for its attributes. The bit assignments are:

31	24	23					0
Reserved		RELOAD					

Table 4-35 SYST_RVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved
[23:0]	RELOAD	Value to load into the SYST_CVR register when the counter is enabled and when it reaches 0, see *Calculating the RELOAD value*

Calculating the RELOAD value

The RELOAD value can be any value in the range `0x00000001-0x00FFFFFF`. A start value of 0 is possible, but has no effect because the SysTick exception request and COUNTFLAG are activated when counting from 1 to 0.

The RELOAD value is calculated according to its use. For example, to generate a multi-shot timer with a period of N processor clock cycles, use a RELOAD value of N-1. If the SysTick interrupt is required every 100 clock pulses, set RELOAD to 99.

4.4.3 SysTick Current Value Register

The SYST_CVR register contains the current value of the SysTick counter. See the register summary in Table 4-33 on page 4-33 for its attributes. The bit assignments are:

| 31	24	23	0
Reserved		CURRENT	
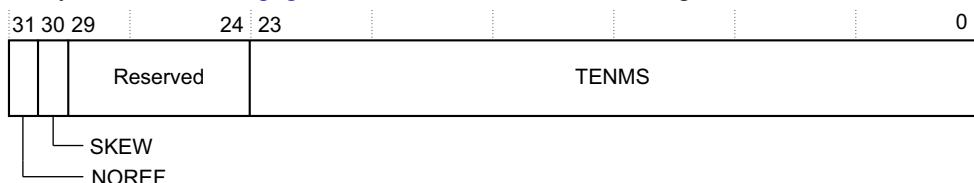
Table 4-36 SYST_CVR bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:0]	CURRENT	Reads return the current value of the SysTick counter. A write of any value clears the field to 0, and also clears the SYST_CSR COUNTFLAG bit to 0.

4.4.4 SysTick Calibration Value Register

This register is optional and might not be available in your implementation.

The SYST_CALIB register indicates the SysTick calibration properties. See the register summary in Table 4-33 on page 4-33 for its attributes. The bit assignments are:

31 30 29	24	23	0
Reserved		TENMS	

SKEW
NOREF

Table 4-37 SYST_CALIB bit assignments

Bits	Name	Function[a]
[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.

a. The functions are implementation defined.

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

4.4.5 SysTick design hints and tips

The SysTick counter runs on the processor clock. If your implementation stops the processor clock signal for low power mode, the SysTick counter stops.

Ensure software uses aligned word accesses to access the SysTick registers.

The SysTick counter reload and current values are optionally undefined at reset, the optional correct initialization sequence for the SysTick counter is:

1. Program reload value.
2. Clear current value.
3. Program Control and Status register.

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

Table 4-38 CMSIS functions for SysTick control

CMSIS SysTick control function	Description
`uint32_t SysTick_Config(uint32_t ticks)`	Creates a periodic SysTick interrupt using the SysTick timer, with a interval defined by the `ticks` parameter

4.5 Processor features

——— **Note** ———

The processor features registers are optional and might not be available in your implementation.

The processor features registers provide software with cache configuration information. The identification space registers are summarized in Table 4-39:

Table 4-39 Identification space summary
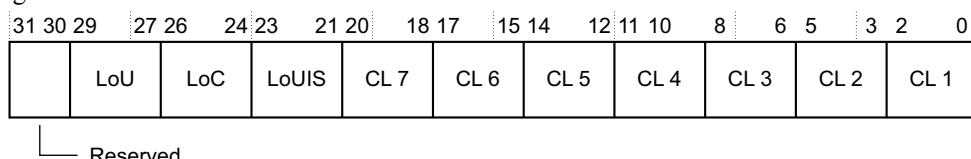
Address	Name	Type	Required privilege	Reset value[a]	Description
0xE000ED78	CLIDR	RO	Privileged	0x09000003	*Cache Level ID Register*
0xE000ED7C	CTR	RO	Privileged	0x8303C003	*Cache Type Register* on page 4-38
0xE000ED80	CCSIDR	RO	Privileged	Unknown	*Cache Size ID Register* on page 4-39r
0xE000ED84	CSSELR	RW	Privileged	Unknown	*Cache Size Selection Register* on page 4-40

a. The reset value is implementation defined.

All registers are only accessible by privileged loads and stores. Unprivileged accesses to these registers result in a BusFault.

4.5.1 Cache Level ID Register

The CLIDR identifies the type of cache, or caches, implemented at each level, and the level of coherency and unification. See the register summary in Table 4-39 for its attributes. The bit assignments are:

Reserved

Table 4-40 CLIDR bit assignments

Bits	Name	Function
[31:30]	-	SBZ.
[29:27]	LoU	Level of Unification: 0b001 Level 2, if either cache is implemented. 0b000 Level 1, if neither instruction nor data cache is implemented.
[26:24]	LoC	Level of Coherency: 0b001 Level 2, if either cache is implemented. 0b000 Level 1, if neither instruction nor data cache is implemented.
[23:21]	LoUIS	RAZ.
[20:18]	CL 7	0b000 No cache at CL 7.
[17:15]	CL 6	0b000 No cache at CL 6.
[14:12]	CL 5	0b000 No cache at CL 5.

Table 4-40 CLIDR bit assignments (continued)

Bits	Name	Function	
[11:9]	CL 4	0b000	No cache at CL 4.
[8:6]	CL 3	0b000	No cache at CL 3.
[5:3]	CL 2	0b000	No cache at CL 2.
[2]	CL 1	**RAZ**	Indicates no unified cache at CL1.
[1]	CL 1	**1**	Data cache is implemented.
		0	No data cache is implemented.
[0]	CL 1	**1**	An instruction cache is implemented.
		0	No instruction cache is implemented.

4.5.2 Cache Type Register

The CTR provides information about the cache architecture. See the register summary in for its attributes. The bit assignments are:

31	29	28	27	24	23	20	19	16	15 14 13	4	3	0
1 0 0		0	CWG		ERG		DMinLine		1 1 0 0 0 0 0 0 0 0 0 0		IminLine	
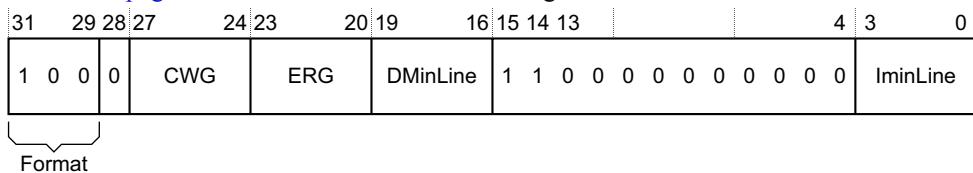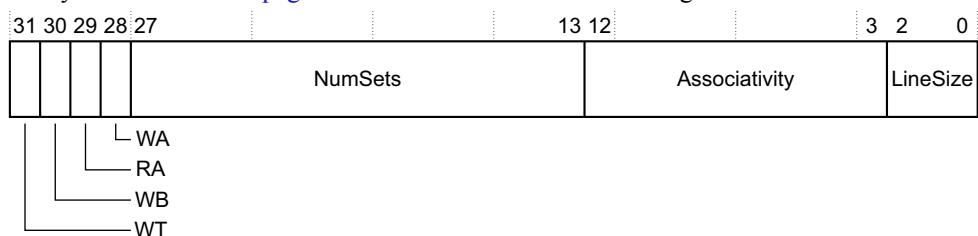
Format

Table 4-41 CTR bit assignments

Bits	Name	Description	
[31:29]	Format	Register format: 0b100 — ARMv7 register format.	
[28]	-	Reserved, RAZ.	
[27:24]	CWG	Cache Writeback Granule: 0b0011 — 8 word granularity for the Cortex-M7 processor.	
[23:20]	ERG	Exclusives Reservation Granule: 0b0000 — The local monitor within the processor does not hold any physical address. It treats any STREX instruction access as matching the address of the previous LDREX instruction. This means that the implemented exclusive reservation granule is the entire memory address range.	
[19:16]	DMinLine	Smallest cache line of all the data and unified caches under the core control: 0b0011 — 8 words for the Cortex-M7 processor.	
[15:14]	-	All bits RAO.	
[13:4]	-	Reserved, RAZ.	
[3:0]	IminLine	Smallest cache line of all the instruction caches under the control of the processor: 0b0011 — 8 words for the Cortex-M7 processor.	

4.5.3 Cache Size ID Register

The CCSIDR identifies the configuration of the cache currently selected by the CSSELR. If no instruction or data cache is configured, the corresponding CCSIDR is RAZ. See the register summary in Table 4-39 on page 4-37 for its attributes. The bit assignments are:


Table 4-42 CCSIDR bit assignments

Bits	Name	Function[a]
[31]	WT	Indicates support available for Write-Through: **1** Write-Through support available.
[30]	WB	Indicates support available for Write-Back: **1** Write-Back support available.
[29]	RA	Indicates support available for read allocation: **1** Read allocation support available.
[28]	WA	Indicates support available for write allocation: **1** Write allocation support available.
[27:13]	NumSets	Indicates the number of sets as: (number of sets) - 1.
[12:3]	Associativity	Indicates the number of ways as: (number of ways) - 1.
[2:0]	LineSize	Indicates the number of words in each cache line.

a. See Table 4-43 on page 4-40 for valid bit field encodings.

The LineSize field is encoded as 2 less than log(2) of the number of words in the cache line. For example, a value of `0x0` indicates there are four words in a cache line, that is the minimum size for the cache. A value of `0x1` indicates there are eight words in a cache line.
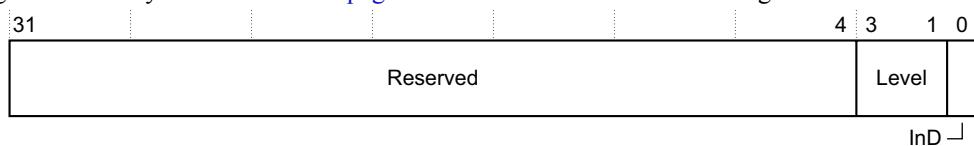
Table 4-43 shows the individual bit field and complete register encodings for the CCSIDR. Use this to determine the cache size for the L1 data or instruction cache selected by the *Cache Size Selection Register* (CSSELR). See *Cache Size Selection Register*.

Table 4-43 CCSIDR encodings

| CSSELR | Cache | Size | Complete register encoding | Register bit field encoding | | | | | | |
				WT	WB	RA	WA	NumSets	Associativity	LineSize
0x0	Data cache	4KB	0xF003E019	1	1	1	1	0x001F	0x3	0x1
		8KB	0xF007E019					0x003F		
		16KB	0xF00FE019					0x007F		
		32KB	0xF01FE019					0x00FF		
		64KB	0xF03FE019					0x01FF		
0x1	Instruction cache	4KB	0xF007E009	1	1	1	1	0x003F	0x1	0x1
		8KB	0xF00FE009					0x007F		
		16KB	0xF01FE009					0x00FF		
		32KB	0xF03FE009					0x01FF		
		64KB	0xF07FE009					0x03FF		

4.5.4 Cache Size Selection Register

The CSSELR selects the cache whose configuration is currently visible in the CCSIDR. See the register summary in Table 4-39 on page 4-37 for its attributes The bit assignments are:

Table 4-44 CSSELR bit assignments

Bit	Name	Description
[31:4]	-	Reserved.
[3:1]	Level	Identifies the cache level selected: 0b000 Level 1 cache. This field is read only, writes are ignored.
[0]	InD	Enables selection of instruction or data cache: **0** Data cache. **1** Instruction cache.

4.5.5 Instruction Error bank Register 0-1

The IEBR0-1 characteristics are:
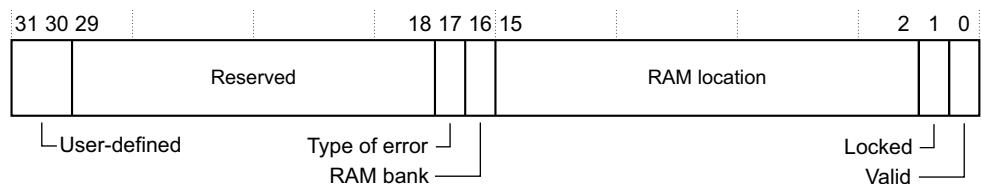
Purpose   Stores information about the error detected in the instruction cache during a cache lookup.

Usage Constraints Accessible in privileged mode only.

Configurations Available if the ECC configurable option is implemented.

Attributes See the register summary in Table 3-1 on page 3-2.

Figure 4-1 shows the IEBR0-1 bit assignments.

Figure 4-1 IEBR0-1 bit assignments

Table 4-45 shows the IEBR0-1 bit assignments.

Table 4-45 IEBR0-1 bit assignments

Bits	Name	Type	Description
[31:30]	-	RW	User-defined. Error detection logic sets this field to `0b00` on a new allocation and on powerup reset.
[29:18]	-	RW	Reserved.
[17]	Type of error	RW	Indicates the error type: **0** Correctable error. **1** Non-correctable error[a].
[16]	RAM bank	RW	Indicates which RAM bank to use: **0** Tag RAM. **1** Data RAM.
[15:2]	RAM location	RW	Indicates the location in instruction cache RAM: **[14]** Way. **[13:4]** Index. **[3:2]** Line doubleword offset.
[1]	Locked	RW	Indicates whether the location is locked or not locked: **0** Location is not locked and available for hardware to allocate. **1** Location is locked by software. Hardware is not allowed to allocate to this entry. Reset by powerup reset to 0.
[0]	Valid	RW	Indicates whether the entry is valid or not: **0** Entry is invalid. **1** Entry is valid. Reset by powerup reset to 0.
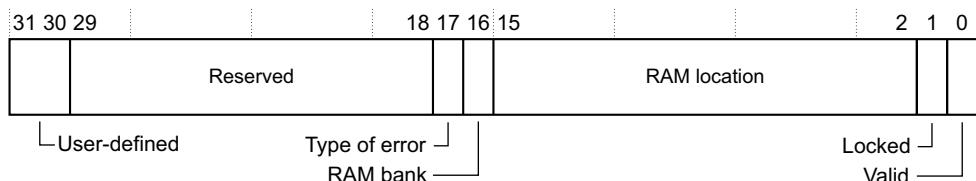
a. Non-correctable errors are recorded when errors are found in multiple bits of the data read from the RAM. These errors result in data loss or data corruption and therefore are non-recoverable.

4.5.6 Data Error bank Register 0-1

The DEBR0-1 characteristics are:

Purpose Stores information about the error detected in the data cache during a cache lookup.

Usage Constraints Accessible in privileged mode only.

Configurations Available if the ECC configurable option is implemented.

Attributes See the register summary in Table 3-1 on page 3-2.

Figure 4-2 shows the DEBR0-1 bit assignments.

Figure 4-2 DEBR0-1

Table 4-46 shows the DEBR0-1 bit assignments.

Table 4-46 DEBR0-1 bit assignments

Bits	Name	Type	Description
[31:30]	-	RW	User-defined. Error detection logic sets this field to 0b00 on a new allocation and on powerup reset.
[29:18]	-	RW	Reserved.
[17]	Type of error	RW	Indicates the error type: **0** Correctable error. **1** Non-correctable error[a].
[16]	RAM bank	RW	Indicates which RAM bank to use: **0** Tag RAM. **1** Data RAM.
[15:2]	RAM location	RW	Indicates the data cache RAM location: **[15:14]** Way. **[13:5]** Index. **[4:2]** Line word offset.
[1]	Locked	RW	Indicates whether the location is locked or not locked: **0** Location is not locked and available for hardware to allocate. **1** Location is locked by software. Hardware is not allowed to allocate to this entry. Reset by powerup reset to 0.
[0]	Valid	RW	Indicates whether the entry is valid or not: **0** Entry is invalid. **1** Entry is valid. Reset by powerup reset to 0.

a. Non-correctable errors are recorded when errors are found in multiple bits of the data read from the RAM. These errors result in data loss or data corruption and therefore are non-recoverable.

4.6 Optional Memory Protection Unit

This section describes the optional *Memory Protection Unit* (MPU).

The MPU divides the memory map into a number of regions, and defines the location, size, access permissions, and memory attributes of each region. It supports:

- Independent attribute settings for each region.
- Overlapping regions.
- Export of memory attributes to the system.

The memory attributes affect the behavior of memory accesses to the region. The Cortex-M7 MPU defines:

- Either 8 or 16 separate memory regions, 0-7 or 0-15 depending on your implementation.
- An optional background region.

When memory regions overlap, a memory access is affected by the attributes of the region with the highest number. For example, the attributes for region 7 take precedence over the attributes of any region that overlaps region 7.

The background region has the same memory access attributes as the default memory map, but is accessible from privileged software only.

The Cortex-M7 MPU memory map is unified. This means instruction accesses and data accesses have same region settings.

If a program accesses a memory location that is prohibited by the MPU, the processor generates a MemManage fault. This causes a fault exception, and might cause termination of the process in an OS environment. 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 *Memory regions, types and attributes* on page 2-12.

Table 4-47 shows the possible MPU region attributes. These include Shareability and cache behavior attributes that are generally only relevant when the processor is configured with caches.

Table 4-47 Memory attributes summary

Memory type	Shareability	Other attributes	Description
Strongly-ordered	-	-	All accesses to Strongly-ordered memory occur in program order. All Strongly-ordered regions are assumed to be shared.
Device	Shared	-	Memory-mapped peripherals that several processors share.
	Non-shared	-	Memory-mapped peripherals that only a single processor uses.
Normal	Shared	Non-cacheable Write-Through Cacheable Write-back Cacheable	Normal memory that is shared between several processors.
	Non-shared	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 MPU registers are:
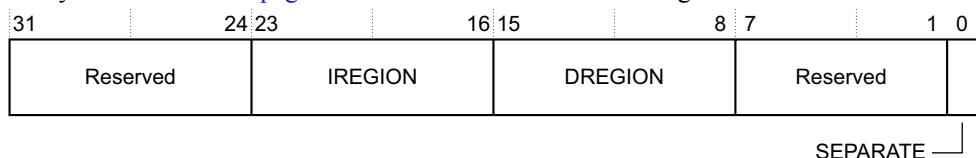
Table 4-48 MPU registers summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000ED90	MPU_TYPE	RO	Privileged	0x00000800	*MPU Type Register*
0xE000ED94	MPU_CTRL	RW	Privileged	0x00000000	*MPU Control Register* on page 4-45
0xE000ED98	MPU_RNR	RW	Privileged	Unknown	*MPU Region Number Register* on page 4-47
0xE000ED9C	MPU_RBAR	RW	Privileged	Unknown	*MPU Region Base Address Register* on page 4-47
0xE000EDA0	MPU_RASR	RW	Privileged	_a	*MPU Region Attribute and Size Register* on page 4-48
0xE000EDA4	MPU_RBAR_A1	RW	Privileged	Unknown	Alias of RBAR, see *MPU Region Base Address Register* on page 4-47
0xE000EDA8	MPU_RASR_A1	RW	Privileged	_a	Alias of RASR, see *MPU Region Attribute and Size Register* on page 4-48
0xE000EDAC	MPU_RBAR_A2	RW	Privileged	Unknown	Alias of RBAR, see *MPU Region Base Address Register* on page 4-47
0xE000EDB0	MPU_RASR_A2	RW	Privileged	_a	Alias of RASR, see *MPU Region Attribute and Size Register* on page 4-48
0xE000EDB4	MPU_RBAR_A3	RW	Privileged	Unknown	Alias of RBAR, see *MPU Region Base Address Register* on page 4-47
0xE000EDB8	MPU_RASR_A3	RW	Privileged	_a	Alias of RASR, see *MPU Region Attribute and Size Register* on page 4-48

a. Unknown apart from the ENABLE field, which is reset to 0.

4.6.1 MPU Type Register

The MPU_TYPE register indicates whether the optional MPU is present, and if so, how many regions it supports. If the MPU is not present the MPU_TYPE register is RAZ. See the register summary in Table 4-48 on page 4-44 for its attributes. The bit assignments are:

31	24	23	16	15	8	7	1	0
Reserved		IREGION		DREGION		Reserved		

SEPARATE ⌐

Table 4-49 TYPE bit assignments
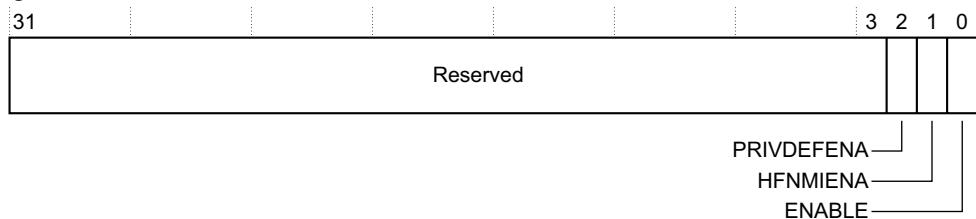
Bits	Name	Function
[31:24]	-	Reserved.
[23:16]	IREGION	Indicates the number of supported MPU instruction regions. Always contains `0x00`. The MPU memory map is unified and is described by the DREGION field.
[15:8]	DREGION	Indicates the number of supported MPU data regions depending on your implementation: `0x08` 8 MPU regions. `0x10` 16 MPU regions.
[7:1]	-	Reserved.
[0]	SEPARATE	Indicates support for unified or separate instruction and date memory maps: `0` Unified.

4.6.2 MPU Control Register

The MPU_CTRL register:

• Enables the optional MPU.

• Enables the optional predefined memory map background region.

• Enables use of the MPU when in the hard fault, *Non-Maskable Interrupt* (NMI), and FAULTMASK escalated handlers.

See the register summary in Table 4-48 on page 4-44 for the MPU_CTRL attributes. The bit assignments are:

| 31	3	2	1	0
Reserved				

PRIVDEFENA ⌐
HFNMIENA ⌐
ENABLE ⌐

Table 4-50 MPU_CTRL bit assignments

Bits	Name	Function	
[31:3]	-	Reserved.	
[2]	PRIVDEFENA	Enables privileged software access to the default memory map:	
		0	If the MPU is enabled, disables use of the default memory map. Any memory access to a location not covered by any enabled region causes a fault.
		1	If the MPU is enabled, enables use of the default memory map as a background region for privileged software accesses.
		When enabled, the background region acts as if it is region number -1. Any region that is defined and enabled has priority over this default map.	
		If the MPU is disabled, the processor ignores this bit.	
[1]	HFNMIENA	Enables the operation of MPU during hard fault, NMI, and FAULTMASK handlers.	
		When the MPU is enabled:	
		0	MPU is disabled during hard fault, NMI, and FAULTMASK handlers, regardless of the value of the ENABLE bit.
		1	The MPU is enabled during hard fault, NMI, and FAULTMASK handlers.
		When the MPU is disabled, if this bit is set to 1 the behavior is Unpredictable.	
[0]	ENABLE	Enables the MPU:	
		0	MPU disabled.
		1	MPU enabled.

When ENABLE and PRIVDEFENA are both set to 1:

- For privileged accesses, the *default memory map* is as described in *Memory model* on page 2-12. Any access by privileged software that does not address an enabled memory region behaves as defined by the default memory map.

- Any access by unprivileged software that does not address an enabled memory region causes a MemManage fault.

XN and Strongly-ordered 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 memory attributes as if the MPU is not implemented, see Table 2-11 on page 2-14. 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 or –2. These priorities are only possible when handling a hard fault or NMI exception, or when FAULTMASK is enabled. Setting the HFNMIENA bit to 1 enables the MPU when operating with these two priorities.

4.6.3 MPU Region Number Register

The MPU_RNR selects which memory region is referenced by the MPU_RBAR and MPU_RASR registers. See the register summary in for its attributes. The bit assignments are:

31	8	7	0
Reserved		REGION	

Table 4-51 MPU_RNR bit assignments
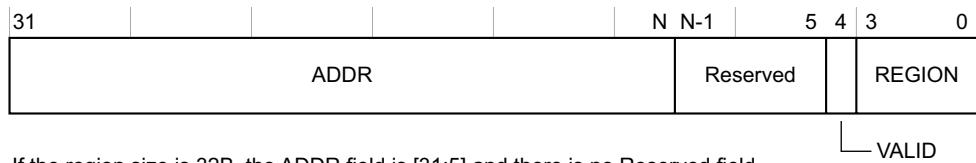
Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	REGION	Indicates the MPU region referenced by the MPU_RBAR and MPU_RASR registers. Depending on your implementation, the MPU supports either 8 or 16 memory regions, so the permitted values of this field are either 0-7 or 0-15.

Normally, you write the required region number to this register before accessing the MPU_RBAR or MPU_RASR. However you can change the region number by writing to the MPU RBAR with the VALID bit set to 1, see *MPU Region Base Address Register*. This write updates the value of the REGION field.

4.6.4 MPU Region Base Address Register

The MPU_RBAR defines the base address of the MPU region selected by the MPU_RNR, and can update the value of the MPU_RNR. See the register summary in for its attributes.

Write MPU_RBAR with the VALID bit set to 1 to change the current region number and update the MPU_RNR. See the register summary in Table 4-48 on page 4-44 for its attributes. The bit assignments are:

If the region size is 32B, the ADDR field is [31:5] and there is no Reserved field

Table 4-52 MPU_RBAR bit assignments

Bits	Name	Function
[31:N]	ADDR	Region base address field. The value of N depends on the region size. For more information see *The ADDR field*.
[(N-1):5]	-	Reserved.

[4]	VALID	MPU Region Number valid bit:
		Write:
		0 MPU_RNR not changed, and depending on your implementation, the processor does one of the following:
		   • Updates the base address for the region specified in the MPU_RNR.
		   • Ignores the value of the REGION field.
		1 The processor, depending on your implementation does one of the following:
		   • Updates the value of the MPU_RNR to the value of the REGION field.
		   • Updates the base address for the region specified in the REGION field.
		Always reads as zero.

[3:0]	REGION	MPU region field: For the behavior on writes, see the description of the VALID field. On reads, returns the current region number, as specified by the RNR.

The ADDR field

The ADDR field is bits[31:N] of the MPU_RBAR. The region size, as specified by the SIZE field in the MPU_RASR, defines the value of N:

$N = Log_2$(Region size in bytes).

If the region size is configured to 4GB, in the MPU_RASR, there is no valid ADDR field. In this case, the region occupies the complete memory map, and the base address is `0x00000000`.

The base address is aligned to the size of the region. For example, a 64KB region must be aligned on a multiple of 64KB, for example, at `0x00010000` or `0x00020000`.

4.6.5 MPU Region Attribute and Size Register

The MPU_RASR defines the region size and memory attributes of the MPU region specified by the MPU_RNR, and enables that region and any subregions. See the register summary in Table 4-48 on page 4-44 for its attributes.

MPU_RASR is accessible using word accesses:

• The most significant halfword holds the region attributes.

- The least significant halfword holds the region size and the region and subregion enable bits.

The bit assignments are:
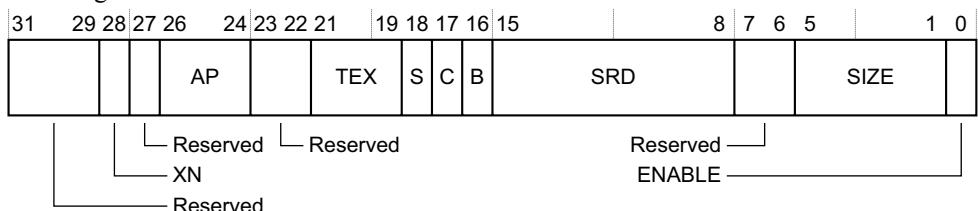
Table 4-53 MPU_RASR bit assignments

Bits	Name	Function
[31:29]	-	Reserved.
[28]	XN	Instruction access disable bit: **0** Instruction fetches enabled. **1** Instruction fetches disabled.
[27]	-	Reserved.
[26:24]	AP	Access permission field, see Table 4-57 on page 4-51.
[23:22]	-	Reserved.
[21:19, 17, 16]	TEX, C, B	Memory access attributes, see Table 4-55 on page 4-50.
[18]	S	Shareable bit, see Table 4-55 on page 4-50.
[15:8]	SRD	Subregion disable bits. For each bit in this field: **0** Corresponding sub-region is enabled. **1** Corresponding sub-region is disabled. See *Subregions* on page 4-53 for more information. Region sizes of 128 bytes and less do not support subregions. When writing the attributes for such a region, write the SRD field as `0x00`.
[7:6]	-	Reserved.
[5:1]	SIZE	Specifies the size of the MPU protection region. The minimum permitted value is 4 (`0b00100`), see *SIZE field values* for more information.
[0]	ENABLE	Region enable bit.

For information about access permission, see *MPU access permission attributes* on page 4-50.

SIZE field values

The SIZE field defines the size of the MPU memory region specified by the RNR as follows:

(Region size in bytes) = $2^{(SIZE+1)}$

The smallest permitted region size is 32B, corresponding to a SIZE value of 4. Table 4-54 gives example SIZE values, with the corresponding region size and value of N in the MPU_RBAR.

Table 4-54 Example SIZE field values

SIZE value	Region size	Value of N[a]	Note
0b00100 (4)	32B	5	Minimum permitted size
0b01001 (9)	1KB	10	-
0b10011 (19)	1MB	20	-
0b11101 (29)	1GB	30	-
0b11111 (31)	4GB	32	Maximum possible size

a. In the MPU_RBAR, see *MPU Region Base Address Register* on page 4-47.

4.6.6 MPU access permission attributes

— **Note** —

Ignore this section if the optional MPU is not present in your implementation.

This section describes the MPU access permission attributes. The access permission bits, TEX, C, B, S, AP, and XN, of the RASR, control access to the corresponding memory region. If an access is made to an area of memory without the required permissions, then the MPU generates a permission fault. Table 4-55 shows encodings for the TEX, C, B, and S access permission bits.

Table 4-55 TEX, C, B, and S encoding

TEX	C	B	S	Memory type	Shareability	Other attributes
0b000	0	0	x[a]	Strongly-ordered	Shareable	-
		1	x[a]	Device	Shareable	-
	1	0	0	Normal	Not shareable	Outer and inner write-through. No write allocate.
			1		Shareable	
		1	0	Normal	Not shareable	Outer and inner write-back. No write allocate.
			1		Shareable	
0b001	0	0	0	Normal	Not shareable	Outer and inner noncacheable.
			1		Shareable	
		1	x[a]	Reserved encoding		-
	1	0	x[a]	Contact the ARM partner for attributes used.		-
		1	0	Normal	Not shareable	Outer and inner write-back. Write and read allocate.
			1		Shareable	

Table 4-55 TEX, C, B, and S encoding (continued)

TEX	C	B	S	Memory type	Shareability	Other attributes
0b010	0	0	x[a]	Device	Not shareable	Nonshared Device.
		1	x[a]	Reserved encoding		-
	1	x[a]	x[a]	Reserved encoding		-
0b1BB	A	A	0	Normal	Not shareable	Cached memory, BB = outer policy, AA = inner policy. See Table 4-56 for the encoding of the AA and BB bits.
			1		Shareable	

    a. The MPU ignores the value of this bit.

Table 4-56 shows the cache policy for memory attribute encodings with a TEX value is in the range 4-7.

Table 4-56 Cache policy for memory attribute encoding

Encoding, AA or BB	Corresponding cache policy
00	Non-cacheable
01	Write back, write and read allocate
10	Write through, no write allocate
11	Write back, no write allocate

Table 4-57 shows the AP encodings that define the access permissions for privileged and unprivileged software.

Table 4-57 AP encoding

AP[2:0]	Privileged permissions	Unprivileged permissions	Description
000	No access	No access	All accesses generate a permission fault
001	RW	No access	Access from privileged software only
010	RW	RO	Writes by unprivileged software generate a permission fault
011	RW	RW	Full access
100	Unpredictable	Unpredictable	Reserved
101	RO	No access	Reads by privileged software only
110	RO	RO	Read only, by privileged or unprivileged software
111	RO	RO	Read only, by privileged or unprivileged software

4.6.7 MPU mismatch

—— **Note** ——

Ignore this section if the optional MPU is not present in your implementation.

When an access violates the MPU permissions, the processor generates a MemManage fault, see *Exceptions and interrupts* on page 2-10. The MMFSR indicates the cause of the fault. See *MemManage Fault Status Register* on page 4-26 for more information.

4.6.8   Updating an MPU region

—— **Note** ——

Ignore this section if the optional MPU is not present in your implementation.

To update the attributes for an MPU region, update the MPU_RNR, MPU_RBAR and MPU_RASR registers. You can program each register separately, or use a multiple-word write to program all of these registers. You can use the MPU_RBAR and MPU_RASR aliases to program up to four regions simultaneously using an STM instruction.

Updating an MPU region using separate words

Simple code to configure one region:

```
; R1 = region number
; R2 = size/enable
; R3 = attributes
; R4 = address
LDR R0,=MPU_RNR          ; 0xE000ED98, MPU region number register
STR R1, [R0, #0x0]       ; Region Number
STR R4, [R0, #0x4]       ; Region Base Address
LSL R3, R3, #16          ; Move attributes to upper half of word
ORR R2, R2, R3           ; Combine attributes and size/enable
STR R2, [R0, #0x8]       ; Update MPU_RASR
```

Disable a region before writing new region settings to the MPU if you have previously enabled the region being changed. For example:

```
; R1 = region number
; R2 = size/enable
; R3 = attributes
; R4 = address
LDR R0,=MPU_RNR          ; 0xE000ED98, MPU region number register
STR R1, [R0, #0x0]       ; Region Number
LDR R5, [R0, #0x8]       ; Read MPU_RASR
BIC R5, R5, #1           ; Clear enable bit
STR R5, [R0, #0x8]       ; Disable region
STR R4, [R0, #0x4]       ; Region Base Address
LSL R3, R3, #16          ; Move attributes to the upper half of the word
ORR R2, R2, R3           ; Combine attributes and size/enable
STR R2, [R0, #0x8]       ; Update MPU_RASR
```

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.

Software does not require any memory barrier instructions during MPU setup, because it accesses the MPU through the PPB, which is a Strongly-ordered memory region.

For example, if you want all of the memory access behavior to take effect immediately after the programming sequence, use a DSB instruction and an ISB instruction. A DSB is required after changing MPU settings, such as at the end of context switch. An ISB is required if the code that

programs the MPU region or regions is entered using a branch or call. If the programming sequence is entered by taking an exception and the programming sequence is exited by using a return from exception then you do not require an ISB instruction.

Updating an MPU region using multi-word writes

You can program directly using multi-word writes, depending on how the information is divided. Consider the following reprogramming:

```
; R1 = region number
; R2 = address
; R3 = size, attributes in one
LDR R0, =MPU_RNR    ; 0xE000ED98, MPU region number register
STR R1, [R0, #0x0]  ; Region Number
STR R2, [R0, #0x4]  ; Region Base Address
STR R3, [R0, #0x8]  ; Region Attribute, Size and Enable
```

Use an STM instruction to optimize this:

```
; R1 = region number
; R2 = address
; R3 = size, attributes in one
LDR R0, =MPU_RNR    ; 0xE000ED98, MPU region number register
STM R0, {R1-R3}     ; Region Number, address, attribute, size and enable
```

You can do this in two words for pre-packed information. This means that the MPU_RBAR contains the required region number and had the VALID bit set to 1, see *MPU Region Base Address Register* on page 4-47. Use this when the data is statically packed, for example in a boot loader:

```
; R1 = address and region number in one
; R2 = size and attributes in one
LDR R0, =MPU_RBAR   ; 0xE000ED9C, MPU Region Base register.
STR R1, [R0, #0x0]  ; Region base address and region number combined
                    ; with VALID (bit 4) set to 1.
STR R2, [R0, #0x4]  ; Region Attribute, Size and Enable.
```
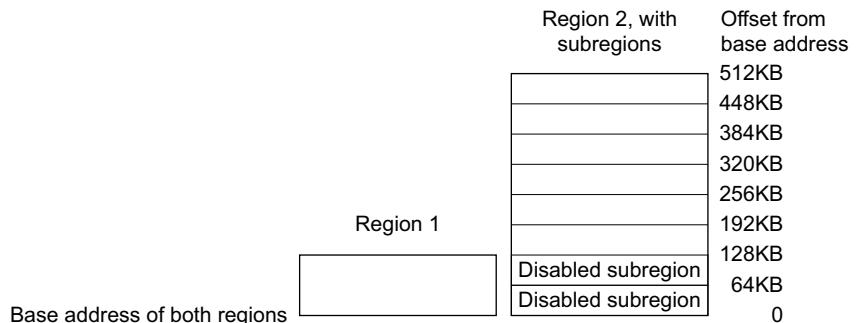
Subregions

Regions of 256 bytes or more are divided into eight equal-sized subregions. Set the corresponding bit in the SRD field of the MPU_RASR to disable a subregion, see *MPU Region Attribute and Size Register* on page 4-48. The least significant bit of SRD controls the first subregion, and the most significant bit controls the last subregion. Disabling a subregion means another region overlapping the disabled range matches instead. If no other enabled region overlaps the disabled subregion, and the access is unprivileged or the background region is disabled, the MPU issues a fault.

Regions of 32, 64, and 128 bytes do not support subregions, With regions of these sizes, you must set the SRD field to `0x00`, otherwise the MPU behavior is Unpredictable.

Example of SRD use

Two regions with the same base address overlap. Region one is 128KB, and region two is 512KB. To ensure the attributes from region one apply to the first 128KB region, set the SRD field for region two to `0b00000011` to disable the first two subregions, as the figure shows.

4.6.9   MPU design hints and tips

—— **Note** ——

Ignore this section if the optional MPU is not present in your implementation.

To avoid unexpected behavior, disable the interrupts before updating the attributes of a region that the interrupt handlers might access.

The processor does not support unaligned accesses to MPU registers.

—— **Note** ——

The MPU registers support aligned word accesses only. Byte and halfword accesses are UNPREDICTABLE.

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.

4.7 Floating Point Unit

This section describes the optional *Floating-Point Unit* (FPU) in a Cortex-M7 device. The FPU implements the FPv5 floating-point extensions.

Depending on your implementation, the FPU fully supports either single-precision, or double-precision, or both for 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.

For implementations with the FPU configured for single-precision only, the FPU contains 32 single-precision extension registers, that you can also access as 16 doubleword registers for load, store, and move operations.

Table 4-58 shows the floating-point system registers in the Cortex-M7 processor with FPU.

Table 4-58 Cortex-M7 floating-point system registers
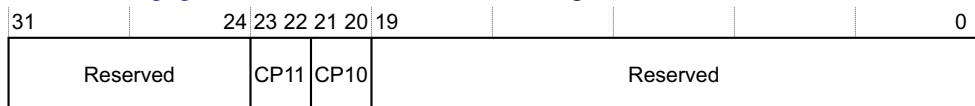
Address	Name	Type	Required privilege	Reset	Description
0xE000ED88	CPACR	RW	Privileged	0x00000000	*Coprocessor Access Control Register* on page 4-56
0xE000EF34	FPCCR	RW	Privileged	0xC0000000	*Floating-point Context Control Register* on page 4-56
0xE000EF38	FPCAR	RW	Privileged	-	*Floating-point Context Address Register* on page 4-57
-	FPSCR[a]	RW	Unprivileged	-	*Floating-point Status Control Register* on page 4-58
0xE000EF3C	FPDSCR	RW	Privileged	0x00000000	*Floating-point Default Status Control Register* on page 4-59

a. The FPSCR register is not memory-mapped, it can be accessed using the VMSR and VMRS instructions, see *VMRS* on page 3-156 and *VMSR* on page 3-157. Software can only access the FPSCR when the FPU is enabled, see *Enabling the FPU* on page 4-59.

The following sections describe the floating-point system registers whose implementation is specific to this processor.

4.7.1 Coprocessor Access Control Register

The CPACR register specifies the access privileges for coprocessors. See the register summary in Table 4-58 on page 4-55 for its attributes. The bit assignments are:

31	24	23 22	21 20	19	0
Reserved		CP11	CP10	Reserved	
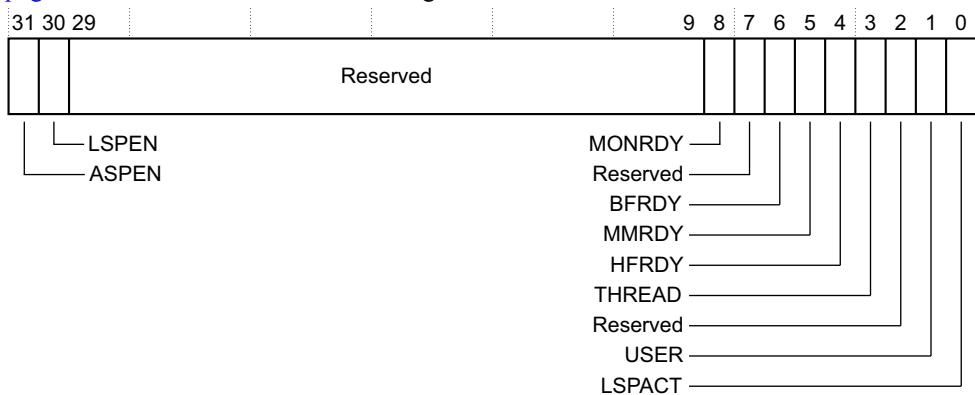
Table 4-59 CPACR bit assignments

Bits	Name	Function
[31:24]	-	Reserved. Read as Zero, Write Ignore.
[2n+1:2n] for n values10 and 11	CPn	Access privileges for coprocessor n. The possible values of each field are: `0b00` Access denied. Any attempted access generates a NOCP UsageFault. `0b01` Privileged access only. An unprivileged access generates a NOCP fault. `0b10` Reserved. The result of any access is Unpredictable. `0b11` Full access.
[19:0]	-	Reserved. Read as Zero, Write Ignore.

4.7.2 Floating-point Context Control Register

The FPCCR register sets or returns FPU control data. See the register summary in Table 4-58 on page 4-55 for its attributes. The bit assignments are:


Table 4-60 FPCCR bit assignments
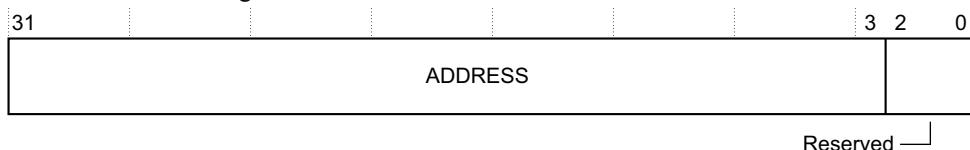
Bits	Name	Function
[31]	ASPEN	Enables CONTROL.FPCA setting on execution of a floating-point instruction. This results in automatic hardware state preservation and restoration, for floating-point context, on exception entry and exit. **0** Disable CONTROL.FPCA setting on execution of a floating-point instruction. **1** Enable CONTROL.FPCA setting on execution of a floating-point instruction.
[30]	LSPEN	**0** Disable automatic lazy state preservation for floating-point context. **1** Enable automatic lazy state preservation for floating-point context.
[29:9]	-	Reserved.

Table 4-60 FPCCR bit assignments (continued)

Bits	Name	Function	
[8]	MONRDY	**0**	DebugMonitor is disabled or priority did not permit setting `MON_PEND` when the floating-point stack frame was allocated.
		1	DebugMonitor is enabled and priority permits setting `MON_PEND` when the floating-point stack frame was allocated.
[7]	-	Reserved.	
[6]	BFRDY	**0**	BusFault is disabled or priority did not permit setting the BusFault handler to the pending state when the floating-point stack frame was allocated.
		1	BusFault is enabled and priority permitted setting the BusFault handler to the pending state when the floating-point stack frame was allocated.
[5]	MMRDY	**0**	MemManage is disabled or priority did not permit setting the MemManage handler to the pending state when the floating-point stack frame was allocated.
		1	MemManage is enabled and priority permitted setting the MemManage handler to the pending state when the floating-point stack frame was allocated.
[4]	HFRDY	**0**	Priority did not permit setting the HardFault handler to the pending state when the floating-point stack frame was allocated.
		1	Priority permitted setting the HardFault handler to the pending state when the floating-point stack frame was allocated.
[3]	THREAD	**0**	Mode was not Thread Mode when the floating-point stack frame was allocated.
		1	Mode was Thread Mode when the floating-point stack frame was allocated.
[2]	-	Reserved.	
[1]	USER	**0**	Privilege level was not user when the floating-point stack frame was allocated.
		1	Privilege level was user when the floating-point stack frame was allocated.
[0]	LSPACT	**0**	Lazy state preservation is not active.
		1	Lazy state preservation is active. Floating-point stack frame has been allocated but saving state to it has been deferred.

4.7.3 Floating-point Context Address Register

The FPCAR register holds the location of the unpopulated floating-point register space allocated on an exception stack frame. See the register summary in Table 4-58 on page 4-55 for its attributes. The bit assignments are:

31	3 2 0
ADDRESS	

Reserved ⌐

Table 4-61 FPCAR bit assignments

Bits	Name	Function
[31:3]	ADDRESS	The location of the unpopulated floating-point register space allocated on an exception stack frame.
[2:0]	-	Reserved. Read as Zero, Writes Ignored.

4.7.4 Floating-point Status Control Register

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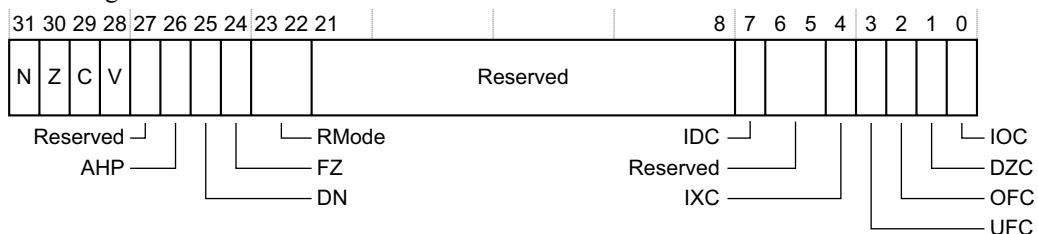

Table 4-62 FPSCR bit assignments

Bits	Name	Function
[31]	N	Condition code flags. Floating-point comparison operations update these flags:
[30]	Z	
[29]	C	
[28]	V	

N	Negative condition code flag.	
Z	Zero condition code flag.	
C	Carry condition code flag.	
V	Overflow condition code flag.	

Bits	Name	Function
[27]	-	Reserved.
[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.
[7]	IDC	Input Denormal cumulative exception bit, see bits [4:0].
[6:5]	-	Reserved.

Bits	Name	Function	
[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.	
[3]	UFC	**IDC, bit[7]**	Input Denormal cumulative exception bit.
[2]	OFC	**IXC**	Inexact cumulative exception bit.
[1]	DZC	**UFC**	Underflow cumulative exception bit.
		OFC	Overflow cumulative exception bit.
[0]	IOC	**DZC**	Division by Zero cumulative exception bit.
		IOC	Invalid Operation cumulative exception bit.

4.7.5 Floating-point Default Status Control Register

The FPDSCR register holds the default values for the floating-point status control data. See the register summary in Table 4-58 on page 4-55 for its attributes. The bit assignments are:
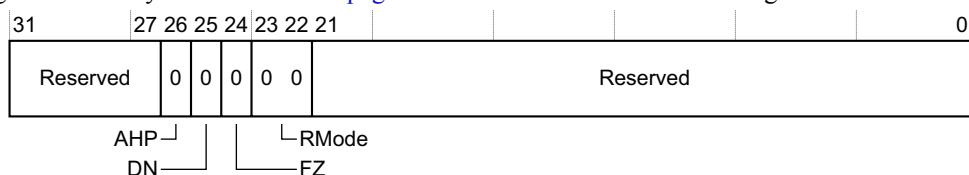

Table 4-63 FPDSCR bit assignments

Bits	Name	Function
[31:27]	-	Reserved
[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

4.7.6 Enabling the FPU

───── **Note** ─────

Ignore this section if the optional FPU is not present in your implementation.

───────────────

The FPU is disabled from reset. You must enable it before you can use any floating-point instructions. Example 4-1 on page 4-60 shows an example code sequence for enabling the FPU in privileged mode. The processor must be in privileged mode to read from and write to the CPACR.

Example 4-1 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.
```

4.8 Cache maintenance operations

───── **Note** ─────

Ignore this section if caches are not present in your implementation.

The cache maintenance operations are only accessible by privileged loads and stores. Unprivileged accesses to these registers always generate a BusFault.

Table 4-64 Cache Maintenance Space register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000EF50	ICIALLU	WO	Privileged	Unknown	Instruction cache invalidate all to the *Point of Unification* (PoU)[a]
0xE000EF54	-	-	-	-	Reserved
0xE000EF58	ICIMVAU	WO	Privileged	Unknown	Instruction cache invalidate by address to the PoU[a]
0xE000EF5C	DCIMVAC	WO	Privileged	Unknown	Data cache invalidate by address to the *Point of Coherency* (PoC)[b]
0xE000EF60	DCISW	WO	Privileged	Unknown	Data cache invalidate by set/way
0xE000EF64	DCCMVAU	WO	Privileged	Unknown	Data cache clean by address to the PoU[a]
0xE000EF68	DCCMVAC	WO	Privileged	Unknown	Data cache clean by address to the PoC[b]
0xE000EF6C	DCCSW	WO	Privileged	Unknown	Data cache clean by set/way
0xE000EF70	DCCIMVAC	WO	Privileged	Unknown	Data cache clean and invalidate by address to the PoC[b]
0xE000EF74	DCCISW	WO	Privileged	Unknown	Data cache clean and invalidate by set/way
0xE000EF78	BPIALL	RAZ/WI	Privileged	-	The BPIALL register is not implemented

a. Cache maintenance operations by PoU can be used to synchronize data between the Cortex-M7 data and instruction Caches, for example when the software uses self-modifying code.

b. Cache maintenance operations by PoC can be used to synchronize data between the Cortex-M7 data cache and an external agent such as a system DMA.

4.8.1 Full instruction cache operation

The ICIALLU is WO and write data is ignored and reads return 0. Writes to this register perform the requested cache maintenance operation. The BPIALL register is not implemented in the Cortex-M7 processor as branch predictor maintenance is not required. The register is RAZ/WI.

4.8.2 Instruction and data cache operations by address

The cache maintenance operations registers are ICIMVAU, DCIMVAC, DCCMVAU, DCCMVAC, and DCCIMVAC. These registers are WO, reads return 0. See the register summary in Table 4-64 for their attributes. The bit assignments are:

Table 4-65 Cache operations registers bit assignments

Bit	Name	Type	Function
[31:5]	MVA	WO	MVA of requested operation

4.8.3 Data cache operations by set-way

The DCISW, DCCSW and DCCISW registers are WO. Reads return 0. See the register summary in Table 4-64 on page 4-61 for their attributes. The bit assignments are:

31 30 29		14 13		5 4	1 0
Way	Reserved		Set	Reserved	0

Table 4-66 Cache operations by set-way bit assignments

Bit	Name	Type	Function
[31:30]	Way	WO	Way that operation applies to. For the data cache, values 0, 1, 2 and 3 are supported.
[29:14]	-	-	Reserved.
[13:5]	Set	WO	Set/index that operation applies to. The number of indices in a cache depends on the configured cache size. When this is less than the maximum, use the LSB of this field. The number of sets in the cache can be determined by reading the *Cache Size ID Register* on page 4-39.
[4:1]	-	-	Reserved.
[0]	-	-	Always reads as zero.

4.8.4 Accessing the Cortex-M7 NVIC cache maintenance operations using CMSIS

CMSIS functions enable software portability between different Cortex-M profile processors. To access cache maintenance operations when using CMSIS, use the following functions:

Table 4-67 CMSIS access cache maintenance operations

CMSIS function	Descriptions
void SCB_EnableICache(void)	Invalidate and then enable instruction cache
void SCB_DisableICache(void)	Disable instruction cache and invalidate its contents
void SCB_InvalidateICache(void)	Invalidate instruction cache
void SCB_EnableDCache(void)	Invalidate and then enable data cache
void SCB_DisableDCache(void)	Disable data cache and then clean and invalidate its contents
void SCB_InvalidateDCache(void)	Invalidate data cache
void SCB_CleanDCache(void)	Clean data cache
void SCB_CleanInvalidateDCache(void)	Clean and invalidate data cache

ARM might add more cache management functions to the CMSIS in the future, and recommends that you check the CMSIS documentation on a regular basis for the latest information.

4.8.5 Initializing and enabling the L1 cache

You can use cache maintenance operations for:

- Cache startup type operations.

- Manipulating the caches so that shared data is visible to other bus masters.

- Enabling data changed by an external DMA agent to be made visible to the Cortex-M7 processor.

After you enable or disable the instruction cache, you must issue an ISB instruction to flush the pipeline. This ensures that all subsequent instruction fetches see the effect of enabling or disabling the instruction cache.

After reset, you must invalidate each cache before enabling it.

When disabling the data cache, you must clean the entire cache to ensure that any dirty data is flushed to external memory.

Before enabling the data cache, you must invalidate the entire data cache if external memory might have changed since the cache was disabled.

Before enabling the instruction cache, you must invalidate the entire instruction cache if external memory might have changed since the cache was disabled.

If your implementation has L1 data and instruction caches, they must be invalidated before they are enabled in software, otherwise UNPREDICTABLE behavior can occur.

Invalidate the entire data cache

Software can use the following code example to invalidate the entire data cache, if it has been included in the processor. The operation is carried out by iterating over each line of the cache and using the DCISW register in the *Private Peripheral Bus* (PPB) memory region to invalidate the line. The number of cache ways and sets is determined by reading the CCSIDR register.

```
CCSIDR  EQU 0xE000ED80
CSSELR  EQU 0xE000ED84
DCISW   EQU 0xE000EF60

        MOV r0, #0x0

        LDR r11, =CSSELR
        STR r0, [r11]                ; Select Data Cache size
        DSB

        LDR r11, =CCSIDR
        LDR r2, [r11]               ; Cache size identification
        AND r1, r2, #0x7            ; Number of words in a cache line
        ADD r7, r1, #0x4
        MOV r1, #0x3ff
        ANDS r4, r1, r2, LSR #3
        MOV r1, #0x7fff
        ANDS r2, r1, r2, LSR #13
        CLZ r6, r4

        LDR r11, =DCISW

inv_loop1
        MOV r1, r4

inv_loop2
        LSL r3, r1, r6
```

```
                  LSL r8, r2, r7
                  ORRr 3, r3, r8

                  STR r3, [r11]             ; Invalidate D-cache line

                  SUBS r1, r1, #0x1
                  BGE inv_loop2
                  SUBS r2, r2, #0x1
                  BGE inv_loop1

                  DSB
                  ISB
```

Invalidate instruction cache

You can use the following code example to invalidate the entire instruction cache, if it has been included in the processor. The operation is carried out by writing to the ICIALLU register in the PPB memory region.

```
ICIALLU EQU 0xE000EF50

                  MOV r0, #0x0
                  LDR r11, =ICIALLU
                  STR r0, [r11]

                  DSB
                  ISB
```

Enabling data and instruction caches

You can use the following code example to enable the data and instruction cache after they have been initialized. The operation is carried out by modifying the CCR.IC and CCR.DC fields in the PPB memory region.

```
CCR     EQU 0xE000ED14

                  LDR r11, =CCR
                  LDR r0, [r11]
                  ORR r0, r0, #0x1:SHL:16    ; Set CCR.DC field
                  ORR r0, r0, #0x1:SHL:17    ; Set CCR.IC field
                  STR r0, [r11]

                  DSB
                  ISB
```

4.8.6 Faults handling considerations

Cache maintenance operations can result in a BusFault. Such fault events are asynchronous.

This type of BusFault:
- Does not cause escalation to HardFault where a BusFault handler is enabled.
- Never causes lockup.

Because the fault event is asynchronous, software code for cache maintenance operations must use memory barrier instructions, such as DSB, on completion so that the fault event can be observed immediately.

4.8.7 Cache maintenance design hints and tips

You must always place a DSB and ISB instruction sequence after a cache maintenance operation to ensure that the effect is observed by any following instructions in the software.

When using a cache maintenance operation by address or set/way a `DSB` instruction must be executed after any previous load or store, and before the maintenance operation, to guarantee that the effect of the load or store is observed by the operation. For example, if a store writes to the address accessed by a DCCMVAC the `DSB` instruction guarantees that the dirty data is correctly cleaned from the data cache.

When one or more maintenance operations have been executed, use of a `DSB` instruction guarantees that they have completed and that any following load or store operations executes in order after the maintenance operations.

Cache maintenance operations always complete in-order with respect to each other. This means only one `DSB` instruction is required to guarantee the completion of a set of maintenance operations.

The following code sequence shows how to use cache maintenance operations to synchronize the data and instruction caches for self-modifying code. The sequence is entered with `<Rx>` containing the new 32-bit instruction. Use `STRH` in the first line instead of `STR` for a 16-bit instruction:

```
STR <Rx>, <inst_address1>
DSB                             ; Ensure the data has been written to the cache.
STR <inst_address1>, DCCMVAU   ; Clean data cache by MVA to point of unification (PoU).
STR <inst_address1>, ICIMVAU   ; Invalidate instruction cache by MVA to PoU.
DSB                             ; Ensure completion of the invalidations.
ISB                             ; Synchronize fetched instruction stream.
```

4.9   Access control

Control of the optional L1 cache ECC and attribute override, the priority of AHB slave traffic, and whether an access is mapped to TCM interfaces or AXI master interface, is defined by the access control registers. The access control registers are:

Table 4-68 Access control register summary
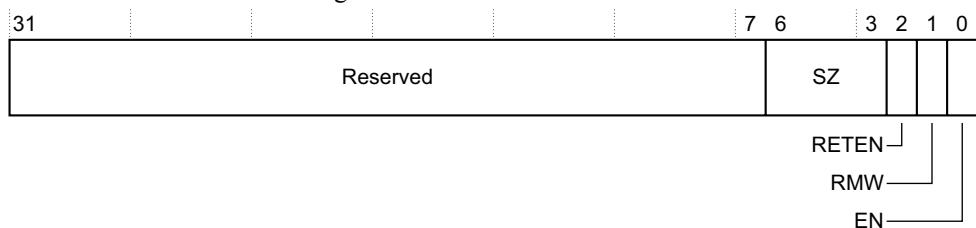
Address	Name	Type	Required privilege	Reset value	Description
0xE000EF90	ITCMCR	RW	Privileged	0x00000000[a]	*Instruction and Data Tightly-Coupled Memory Control Registers* on page 4-67
0xE000EF94	DTCMCR	RW	Privileged	0x00000000[a]	
0xE000EF98	AHBPCR	RW	Privileged	0x00000000[a]	*AHBP Control Register* on page 4-69
0xE000EF9C	CACR	RW	Privileged	-[b]	*L1 Cache Control Register* on page 4-70
0xE000EFA0	AHBSCR	RW	Privileged	0x00000800	*AHB Slave Control Register* on page 4-72
0xE000EFA8	ABFSR	RW	Privileged	0x00000000	*Auxiliary Bus Fault Status register* on page 4-73

a. Implementation defined.

b. The reset value is implementation and configuration dependent and the silicon vendor changes this. If cache ECC is configured the reset value is 0x00000000, if cache ECC is not configured the reset value is 0x00000002.

4.9.1 Instruction and Data Tightly-Coupled Memory Control Registers

The ITCMCR and DTCMCR control whether access is mapped to the TCM interfaces or the AXI master interface. The bit assignments are:


Table 4-69 ITCMCR and DTCMCR bit assignments

Bits	Name	Type	Function
[31:7]	-	-	Reserved, RAZ/WI.
[6:3]	SZ	RO	TCM size. Indicates the size of the relevant TCM. The values that apply depend on your implementation: 0b0000 No TCM implemented. 0b0011 4KB. 0b0100 8KB. 0b0101 16KB. 0b0110 32KB. 0b0111 64KB. 0b1000 128KB. 0b1001 256KB. 0b1010 512KB. 0b1011 1MB. 0b1100 2MB. 0b1101 4MB. 0b1110 8MB. 0b1111 16MB. All other encodings are reserved.
[2]	RETEN[a]	RW	Retry phase enable. When enabled the processor guarantees to honor the retry output on the corresponding TCM interface: **0** Retry phase disabled. **1** Retry phase enabled.
[1]	RMW[b]	RW	*Read-Modify-Write* (RMW) enable. Indicates that all sub-chunk writes to a given TCM use a RMW sequence: **0** RMW disabled. **1** RMW enabled.
[0]	EN	RW	TCM enable. When a TCM is disabled all accesses are made to the AXI master. **0** TCM disabled. **1** TCM enabled.

a. The RETEN field in the ITCMCR and DTCMCR is used to support the optional error detection and correction in the TCM.

b. The RMW field in the ITCMCR and DTCMCR is used to support the optional error detection and correction in the TCM.

Enabling the TCM

The TCM interfaces can be enabled at reset in the system by an external signal on the processor. If they are disabled at reset then the following code example can be used to enable both the instruction and data TCM interfaces in software:

```
ITCMCR  EQU 0xE000EF90
DTCMCR  EQU 0xE000EF94

        LDR r11, =ITCMCR
        LDR r0, [r11]
        ORR r0, r0, #0x1    ; Set ITCMCR.EN field
        STR r0, [r11]

        LDR r11, =DTCMCR
        LDR r0, [r11]
        ORR r0, r0, #0x1    ; Set DTCMCR.EN field
        STR r0, [r11]

        DSB
        ISB
```

Enabling the TCM retry and read-modify-write

If the TCM connected to the processor supports error detection and correction, the TCM interface must be configured to support the retry and read-modify-write features. These can be enabled at reset in the system by external signals on the processor. If they are disabled at reset then the following code example can be used to enable them in software:

```
ITCMCR  EQU 0xE000EF90
DTCMCR  EQU 0xE000EF94

        LDR r11, =ITCMCR
        LDR r0, [r11]
        ORR r0, r0, #0x1:SHL:1    ; Set ITCMCR.RMW field
        ORR r0, r0, #0x1:SHL:2    ; Set ITCMCR.RETEN field
        STR r0, [r11]

        LDR r11, =DTCMCR
        LDR r0, [r11]
        ORR r0, r0, #0x1:SHL:1    ; Set DTCMCR.RMW field
        ORR r0, r0, #0x1:SHL:2    ; Set DTCMCR.RETEN field
        STR r0, [r11]

        DSB
        ISB
```
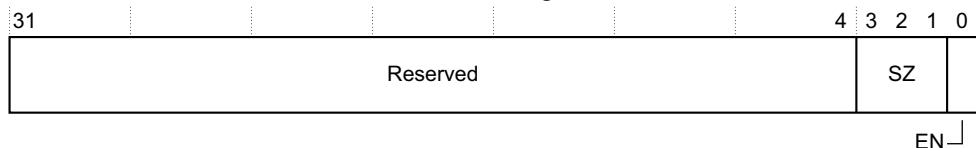
4.9.2   AHBP Control Register

> ——— **Note** ———
>
> This register is optional and might not be available in your implementation.

Depending on your implementation, the AHBPCR controls accesses to the device either on the AHBP interface or AXI master interface. The bit assignments are:

31	4	3	2	1	0
Reserved		SZ			

EN⌐

Table 4-70 AHBPCR bit assignments

Bits	Name	Type	Function
[31:4]	-	-	Reserved, RAZ/WI.
[3:1]	SZ	RO	AHBP size. The values that apply depend on your implementation: 0b001 64MB. 0b010 128MB. 0b011 256MB. 0b100 512MB.
[0]	EN	RW	AHBP enable: 0 AHBP disabled. When disabled all accesses are made to the AXI master. 1 AHBP enabled.

Enabling the AHBP interface

Depending on your implementation, the AHBP interface can be enabled at reset in the system by an external signal on the processor. If it is disabled at reset then the following code example can be used to enable the AHPB interface from software:

```
AHBPCR  EQU 0xE000EF98

        LDR r11, =AHBPCR
        LDR r0, [r11]
        ORR r0, r0, #0x1        ; Set AHBPCR.EN field
        STR r0, [r11]

        DSB
        ISB
```
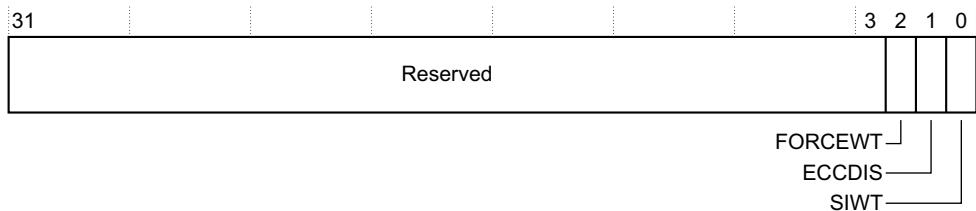
4.9.3 L1 Cache Control Register

—— **Note** ——

This register is optional and might not be available in your implementation.

The CACR controls the optional L1 ECC and the L1 cache coherency usage model. The bit assignments are:

Table 4-71 CACR bit assignments

Bits	Name	Type	Function
[31:3]	-	-	Reserved, RAZ/WI.
[2]	FORCEWT	RW	Enables Force Write-Through in the data cache: **0** Disables Force Write-Through. **1** Enables Force Write-Through. All Cacheable memory regions are treated as Write-Through. This bit is RAZ/WI if the data cache is excluded. If the data cache is included the reset value of FORCEWT is 0.
[1]	ECCDIS	RW	Enables ECC in the instruction and data cache: **0** Enables ECC in the instruction and data cache. This is RAO/WI if both data cache and instruction cache are excluded or if ECC is excluded. **1** Disables ECC in the instruction and data cache. This is RAZ/WI if data cache is not configured.
[0]	SIWT	RW	Shared cacheable-is-WT for data cache. Enables cache coherency usage: **0** Normal Cacheable Shared locations are treated as being Non-cacheable. Programmed inner cacheability attributes are ignored. This is the default mode of operation for Shared memory. The data cache is transparent to software for these locations and therefore no software maintenance is required to maintain coherency. **1** Normal Cacheable shared locations are treated as Write-Through. Programmed inner cacheability attributes are ignored. All writes are globally visible. Other memory agent updates are not visible to Cortex-M7 processor software without suitable cache maintenance. Useful for heterogeneous MP-systems where, for example, the Cortex-M7 processor is integrated on the *Accelerator Coherency Port* (ACP) interface on an MP-capable processor. This bit is RAZ/WI when data cache is not configured.

Disabling cache error checking and correction

If cache error checking and correction is included in the processor it is enabled by default from reset. The following code example can be used to disable the feature. The operation is carried out by modifying the CACR.ECCEN field in the PPB memory region.

```
CACR    EQU 0xE000EF9C
        LDR r11, =CACR
        LDR r0, [r11]
```

```
        BFC r0, #0x1, #0x1    ; Clear CACR.ECCEN
        STR r0, [r11]

        DSB
        ISB
```

Care must be taken when software changes CACR.ECCEN. If CACR.ECCEN changes when the caches contain data, ECC information in the caches might not be correct for the new setting, resulting in unexpected errors and data loss. Therefore software must only change CACR.ECCEN when both caches are turned off and both caches must be invalidated after the change.

4.9.4 AHB Slave Control Register

> ── **Note** ──
>
> This register is optional and might not be available in your implementation.

The AHBSCR is used by software to control the priority of AHB slave traffic. The bit assignments are:
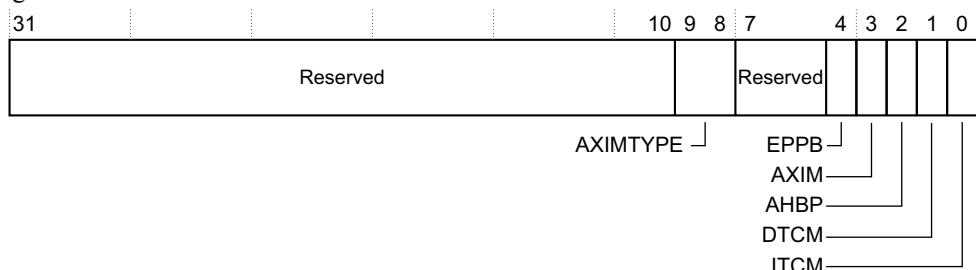
31		15	11 10		2 1 0
Reserved		INITCOUNT		TPRI	CTL

Table 4-72 AHBSCR bit assignments

Bits	Name	Type	Function
[31:16]	-	-	Reserved.
[15:11]	INITCOUNT	RW	Fairness counter initialization value. Use to demote access priority of the requestor selected by the AHBSCR.CTL field. The reset value is `0b01`. ── **Note** ── • For round-robin mode set INITCOUNT to `0b01` and AHBSCR.CTL to `0b00` or `0b01`. • INITCOUNT must not be set to `0b00` because the demoted requestor always takes priority when contention occurs, which can lead to livelock. • INITCOUNT is not used when AHBSCR.CTL is `0b11`.
[10:2]	TPRI	RW	Threshold execution priority for AHBS traffic demotion. `0b0xxxxxxx` Priority is TPRI[7:0]. This is the same as the NVIC register encodings. `0b11111111` Priority of -1. This is the priority of the HardFault exception. `0b11111110` Priority of -2. This is the priority of the NMI exception.
[1:0]	CTL	RW	AHBS prioritization control: `0b00` AHBS access priority demoted. This is the reset value. `0b01` Software access priority demoted. `0b10` AHBS access priority demoted by initializing the fairness counter to the AHBSCR.INITCOUNT value when the software execution priority is higher than or equal to the threshold level programmed in AHBSCR.TPRI. When the software execution priority is below this value, the fairness counter is initialized with 1 (round-robin). ── **Note** ── The threshold level encoding matches the NVIC encoding and uses arithmetically larger numbers to represent lower priority. `0b11` **AHBSPRI** signal has control of access priority.

4.9.5 Auxiliary Bus Fault Status register

The ABFSR stores information on the source of asynchronous bus faults. The ASBFSR bit assignments are:

Table 4-73 ABFSR bit assignments

Bits	Name	Function
[31:10]	-	Reserved.
[9:8]	AXIMTYPE	Indicates the type of fault on the AXIM interface: 0b00 OKAY. 0b01 EXOKAY. 0b10 SLVERR. 0b11 DECERR. Only valid when AXIM is 1.
[7:5]	-	Reserved.
[4]	EPPB	Asynchronous fault on EPPB interface.[a]
[3]	AXIM	Asynchronous fault on AXIM interface.[a]
[2]	AHBP	Asynchronous fault on AHBP interface[a].
[1]	DTCM	Asynchronous fault on DTCM interface.[a]
[0]	ITCM	Asynchronous fault on ITCM interface.[a]

a. This interface might not be present in your implementation.

In the bus-fault handler, software reads the BFSR, and if an asynchronous fault occurs, the ABFSR is read to determine which interfaces are affected. The ABFSR[4:0] fields remains valid until cleared by writing to the ABFSR with any value.

For more information about the BFSR, see *BusFault Status Register* on page 4-27.

Appendix A
Cortex-M7 Options

This appendix describes the configuration options for a Cortex-M7 processor implementation. It shows what features of a Cortex-M7 implementation are determined by the device manufacturer. It contains the following section:

A.1 Cortex-M7 processor options

Table A-1 shows the Cortex-M7 processor implementation options.

Table A-1 Effects of the Cortex-M7 processor implementation options

Option	Description, and affected documentation
RTL version	The RTL version can affect the availability of some features. This affects: • Variant and Revision field values in Table 4-14 on page 4-13. • The CPUID register reset value in Table 4-12 on page 4-11.
Inclusion of MPU	The implementer decides whether to include the *Memory Protection Unit* (MPU). This affects references to the MPU, or MPU registers in: • *About the Cortex-M7 processor and core peripherals* on page 1-2. • *Memory regions, types and attributes* on page 2-12. • *Behavior of memory accesses* on page 2-14, after Table 2-11 on page 2-14. • *Software ordering of memory accesses* on page 2-15. • *Exception types* on page 2-19, in the description of *MemManage fault.* • *Fault handling* on page 2-27. • The MPU mismatch entries in Table 2-16 on page 2-27, except for the IACCVIOL entry. • Footnote ᵃ to Table 2-16 on page 2-27. • Table 4-1 on page 4-2. When the processor does not include an MPU, the Memory Protection Unit row address range must change to `0xE000ED90-0xE000ED93`, MPU Type Register, reads as zero, and the following footnote applies: — Software can read the MPU Type Register at `0xE000ED90` to test for the presence of a *Memory Protection Unit* (MPU). • Table 4-25 on page 4-24, in the description of the IACCVIOL bit. The MPU can be implemented either as an 8-region MPU or a 16-region MPU. This affects the DREGION bits in the MPU_TYPE register.
Inclusion of FPU	The implementer decides whether to include the *Floating-Point Unit* (FPU). The FPU implemented can be a single-precision only FPU, or single-precision and double-precision FPU. This affects: • *Floating-point instructions* on page 3-135. • Interruptible-continuable instructions in *Core registers* on page 2-3. • The FPCA bit in the CONTROL register. • Table 2-15 on page 2-26. • The MLSPERR bit in the *MemManage Fault Status Register* (MMFSR). • The LSPERR bit in the *BusFault Status Register* (BFSR).
Number of interrupts	The implementer decides how many interrupts the Cortex-M7 processor implementation supports, in the range 1-240. This affects: • The maximum value of ISR_NUMBER in Table 2-5 on page 2-6. • Entries in the last row of Table 2-14 on page 2-20, particularly if only one interrupt is implemented. • The maximum interrupt number, and associated information where appropriate, in: — *Exception handlers* on page 2-21. — Figure 2-1 on page 2-22. — *Nested Vectored Interrupt Controller* on page 4-3. • The number of implemented *Nested Vectored Interrupt Controller* (NVIC) registers in: — Table 4-2 on page 4-3. — The appropriate register descriptions in sections *Interrupt Set-enable Registers* on page 4-4 to *Interrupt Priority Registers* on page 4-7. • *Vector Table Offset Register* on page 4-17, including the figure and Table 4-16 on page 4-17. See the configuration information in the section for guidance on the required configuration.

Table A-1 Effects of the Cortex-M7 processor implementation options (continued)

Option	Description, and affected documentation
Number of priority bits	The implementer decides how many priority bits are implemented in priority value fields, in the range 3-8. Register priority value fields are 8 bits wide, and un-implemented low-order bits read as zero and ignore writes. This affects: • The footnote to *Table 2-9 on page 2-9*. • The maximum priority level value in the introduction to *Nested Vectored Interrupt Controller* on page 4-3. • In *Interrupt Priority Registers* on page 4-7: — The maximum priority level value, in the introductory sentence. — The priority field description, in *Table 4-9 on page 4-7*. • In *System Handler Priority Registers* on page 4-22: — The field width, in the introductory sentence. — The priority fields description, after *Table 4-21 on page 4-22*. • The description of the effect of the binary point, in *Binary point on page 4-19*.
Inclusion of the WIC	The implementer decides whether to include the *Wakeup Interrupt Controller* (WIC). This affects: • References to the WIC in *About the Cortex-M7 processor and core peripherals on page 1-2*. • The description of deep sleep mode and the selection of the sleep mode in the introduction to the section *Power management on page 2-30*. • *The optional Wakeup Interrupt Controller on page 2-31*.
Sleep mode power-saving	The implementer decides what power-saving sleep modes to implement, see *Power management* on page 2-30. Sleep mode power saving might also affect SysTick behavior, see *SysTick design hints and tips on page 4-36*.
Register reset values	The implementer decides whether all registers in the register bank can be reset. This affects the reset values in *Table 2-2 on page 2-3*.
Endianness	The implementer decides whether the memory system is little-endian or big-endian, and is dependent on the external signal, **CFGBIGEND**, see *Data types on page 2-10*, *Memory endianness on page 2-15* and ENDIANNESS bit in *Table 4-17 on page 4-18*.

Table A-1 Effects of the Cortex-M7 processor implementation options (continued)

Option	Description, and affected documentation
Memory features	Some features of the memory system are implementation specific. This affects details of vendor-specific memory in *Memory model* on page 2-12, including: • The figure in that section. The memory map configuration of the Code SRAM and peripheral memory region 0x00000000 to 0x5FFFFFFF to any implemented TCM and system space. • Information in Table 2-11 on page 2-14. • The additional memory attributes available. • Information in Table 2-12 on page 2-14. The implementer decides whether to include caches. This affects references to caches or cache registers in: • *About the Cortex-M7 processor and core peripherals* on page 1-2. • *Processor features* on page 4-37. • *Cache maintenance operations* on page 4-61. The implementer decides whether to include *Tightly-Coupled Memories* (TCMs). This affects references to *Tightly-Coupled Memory* (TCM) and registers in: • *About the Cortex-M7 processor and core peripherals* on page 1-2. • *Access control* on page 4-66.
SysTick SYST_CALIB register	The implementation of this register is implementation defined. This affects: • *SysTick Calibration Value Register* on page 4-35. • The entry for SYST_CALIB in Table 4-33 on page 4-33.
VTOR.TBLOFF[31:7] vector base address	The initial value in the *Vector Table Offset Register* (VTOR), which controls the vector base address, is implementation defined, and is dependent on the external signal, **INITVTOR**. This affects the configuration of the TBLOFF[31:7] bit field in Table 4-16 on page 4-17. This also affects the address from where the processor loads: • The MSP value in *Stack Pointer* on page 2-4. • The PC value in *Program Counter* on page 2-4.

Appendix B
Revisions

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

Table B-1 Issue A

Change	Location	Affects
First release	-	-

Glossary

This glossary describes some of the terms used in technical documents from ARM.

Abort
A mechanism that indicates to a processor that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory.

Aligned
A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

Banked register
A register that has multiple physical copies, where the state of the processor determines which copy is used. The Stack Pointer, SP (R13) is a banked register.

Base register
In instruction descriptions, a register specified by a load or store instruction that is used to hold the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the address that is sent to memory.

See also Index register.

Big-endian (BE)
Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Byte-invariant, Endianness, Little-endian (LE).

Big-endian memory
Memory in which:

* a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address

- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

See also Little-endian memory.

Breakpoint A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

Byte-invariant In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access. An ARM byte-invariant implementation also supports unaligned halfword and word memory accesses. It expects multi-word accesses to be word-aligned.

Cache A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions, data, or instructions and data. This is done to greatly increase the average speed of memory accesses and so improve processor performance.

Condition field A four-bit field in an instruction that specifies a condition under which the instruction can execute.

Conditional execution

If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Context The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.

Coprocessor A processor that supplements the main processor. The Cortex-M7 processor does not support any coprocessors.

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Direct Memory Access (DMA)

An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

Doubleword A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

Doubleword-aligned

A data item having a memory address that is divisible by eight.

Endianness Byte ordering. The scheme that determines the order that successive bytes of a data word are stored in memory. An aspect of the systems memory mapping.

See also Little-endian and Big-endian.

Exception An event that interrupts program execution. When an exception occurs, the processor suspends the normal program flow and starts execution at the address indicated by the corresponding exception vector. The indicated address contains the first instruction of the handler for the exception.

An exception can be an interrupt request, a fault, or a software-generated system exception. Faults include attempting an invalid memory access, attempting to execute an instruction in an invalid processor state, and attempting to execute an undefined instruction.

Exception service routine

See Interrupt handler.

Exception vector *See* Interrupt vector.

Flat address mapping

A system of organizing memory in which each physical address in the memory space is the same as the corresponding virtual address.

Halfword A 16-bit data item.

Illegal instruction An instruction that is architecturally Undefined.

Implementation defined

The behavior is not architecturally defined, but is defined and documented by individual implementations.

Implementation specific

The behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

Index register In some load and store instruction descriptions, the value of this register is used as an offset to be added to or subtracted from the base register value to form the address that is sent to memory. Some addressing modes optionally enable the index register value to be shifted prior to the addition or subtraction.

See also Base register.

Instruction cycle count

The number of cycles that an instruction occupies the Execute stage of the pipeline.

Interrupt handler A program that control of the processor is passed to when an interrupt occurs.

Interrupt vector One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

Little-endian (LE) Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian (BE), Byte-invariant, Endianness.

Little-endian memory

Memory in which:

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

- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

See also Big-endian memory.

Load/store architecture

A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.

Memory Protection Unit (MPU)

Hardware that controls access permissions to blocks of memory. An MPU does not perform any address translation.

Prefetching

In pipelined processors, the process of fetching instructions from memory to fill up the pipeline before the preceding instructions have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Read

Reads are defined as memory operations that have the semantics of a load. Reads include the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP.

Region

A partition of memory space.

Reserved

A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Should Be One (SBO)

Write as 1, or all 1s for bit fields, by software. Writing as 0 produces Unpredictable results.

Should Be Zero (SBZ)

Write as 0, or all 0s for bit fields, by software. Writing as 1 produces Unpredictable results.

Should Be Zero or Preserved (SBZP)

Write as 0, or all 0s for bit fields, by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

Thread-safe

In a multi-tasking environment, thread-safe functions use safeguard mechanisms when accessing shared resources, to ensure correct operation without the risk of shared access conflicts.

Thumb instruction

One or two halfwords that specify an operation for a processor to perform. Thumb instructions must be halfword-aligned.

Unaligned

A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

Undefined

Indicates an instruction that generates an Undefined instruction exception.

Unpredictable (UNP)

You cannot rely on the behavior. Unpredictable behavior must not represent security holes. Unpredictable behavior must not halt or hang the processor, or any parts of the system.

Warm reset

Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

WA

See Write-allocate (WA).

WB

See Write-back (WB).

Word

A 32-bit data item.

Write

Writes are defined as operations that have the semantics of a store. Writes include the Thumb instructions STM, STR, STRH, STRB, and PUSH.

Write-allocate (WA)

In a write-allocate cache, a cache miss on storing data causes a cache line to be allocated into the cache.

Write-back (WB) In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. This is also known as copyback.

Write buffer A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.

Write-through (WT) In a write-through cache, data is written to main memory at the same time as the cache is updated.