Cortex-M4 Devices
Generic User Guide

Copyright © 2010 ARM. All rights reserved.

Release Information

The following changes have been made to this book.

Change history

<table>
<thead>
<tr>
<th>Date</th>
<th>Issue</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 December 2010</td>
<td>A</td>
<td>Non-Confidential</td>
<td>First release</td>
</tr>
</tbody>
</table>

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

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

Cortex-M4 Devices Generic User Guide

Preface
About this book ........................................................................................................... vi
Feedback .................................................................................................................... ix

Chapter 1  Introduction
1.1 About the Cortex-M4 processor and core peripherals ........................................ 1-2

Chapter 2  The Cortex-M4 Processor
2.1 Programmers model ............................................................................................ 2-2
2.2 Memory model .................................................................................................. 2-12
2.3 Exception model ............................................................................................... 2-21
2.4 Fault handling .................................................................................................. 2-29
2.5 Power management ......................................................................................... 2-32

Chapter 3  The Cortex-M4 Instruction Set
3.1 Instruction set summary ................................................................................... 3-2
3.2 CMSIS functions ............................................................................................. 3-9
3.3 About the instruction descriptions ................................................................... 3-11
3.4 Memory access instructions ........................................................................... 3-22
3.5 General data processing instructions ............................................................. 3-39
3.6 Multiply and divide instructions .................................................................... 3-74
3.7 Saturating instructions .................................................................................. 3-95
3.8 Packing and unpacking instructions ............................................................... 3-107
3.9 Bitfield instructions ....................................................................................... 3-114
3.10 Branch and control instructions ................................................................... 3-118
3.11 Floating-point instructions .......................................................................... 3-126
3.12 Miscellaneous instructions ........................................................................... 3-157
Chapter 4  Cortex-M4 Peripherals
4.1  About the Cortex-M4 peripherals ................................................................. 4-2
4.2  Nested Vectored Interrupt Controller ............................................................. 4-3
4.3  System control block ..................................................................................... 4-11
4.4  System timer, SysTick .................................................................................. 4-33
4.5  Optional Memory Protection Unit ................................................................. 4-37
4.6  Floating Point Unit (FPU) ............................................................................ 4-48

Appendix A  Cortex-M4 Options
A.1  Cortex-M4 implementation options ............................................................. A-2

Glossary
Preface

This preface introduces the Cortex-M4 Devices Generic User Guide. It contains the following sections:

• About this book on page vi
• Feedback on page ix.
About this book

This book is a generic user guide for devices that implement the ARM Cortex-M4 processor. Implementers of Cortex-M4 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
- some features are described as optional.

In this book, unless the context indicates otherwise:

- **Processor** Refers to the Cortex-M4 processor, as supplied by ARM.
- **Device** Refers to an implemented device, supplied by an ARM partner, that incorporates a Cortex-M4 processor. In particular, your device refers to the particular implementation of the Cortex-M4 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 `rmpn` identifier indicates the revision status of the product described in this book, where:

- **rn** Identifies the major revision of the product.
- **pn** Identifies the minor revision or modification status of the product.

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-M4 processor.

Using this book

This book is organized into the following chapters:

- **Chapter 1 Introduction**
  Read this for an introduction to the Cortex-M4 processor and its features.

- **Chapter 2 The Cortex-M4 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-M4 Instruction Set**
  Read this for information about the processor instruction set.

- **Chapter 4 Cortex-M4 Peripherals**
  Read this for information about Cortex-M4 peripherals.

- **Appendix A Cortex-M4 Options**
  Read this for information about the processor implementation and configuration options.

- **Glossary**
  Read this for definitions of terms used in this book.
Typographical conventions

The typographical conventions used in this document are:

- **italic**
  - Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

- **bold**
  - Used for terms in descriptive lists, where appropriate.

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

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

- `< and >`
  - Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example:
    
    ```
    OMP Rn, <Rm|#imm>
    ```
Additional reading

This section lists publications by ARM and by third parties.


See onARM, http://onarm.com, 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:


**Other publications**

This guide only provides generic information for devices that implement the ARM Cortex-M4 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 0553A
• the page numbers to which your comments apply
• a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.
Chapter 1

Introduction

This chapter introduces the Cortex-M4 processor and its features. It contains the following section:

- About the Cortex-M4 processor and core peripherals on page 1-2.
1.1 About the Cortex-M4 processor and core peripherals

The Cortex-M4 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).

The Cortex-M4 processor is built on a high-performance processor core, with a 3-stage pipeline Harvard architecture, making it ideal for demanding embedded applications. The processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design, providing high-end processing hardware including optional IEEE754-compliant single-precision floating-point computation, a range of single-cycle and SIMD multiplication and multiply-with-accumulate capabilities, saturating arithmetic and dedicated hardware division.

To facilitate the design of cost-sensitive devices, the Cortex-M4 processor implements tightly-coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex-M4 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-M4 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-M4 processor closely integrates a configurable Nested Vectored Interrupt Controller (NVIC), to deliver industry-leading interrupt performance. The NVIC includes a Non Maskable Interrupt (NMI) that 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.

1.1.1 System-level interface

The Cortex-M4 processor provides multiple interfaces using AMBA® technology to provide high speed, low latency memory accesses. It supports unaligned data accesses and implements atomic bit manipulation that enables faster peripheral controls, system spinlocks and thread-safe Boolean data handling.

The Cortex-M4 processor has an optional Memory Protection Unit (MPU) that permits control of individual regions in memory, 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 Optional integrated configurable debug

The Cortex-M4 processor can implement 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.

For system trace the processor integrates an Instrumentation Trace Macrocell (ITM) alongside 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 Embedded Trace Macrocell™ (ETM) delivers unrivalled instruction 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 optional Flash Patch and Breakpoint Unit (FPB) provides up to eight hardware breakpoint comparators that debuggers can use. The comparators in the FPB also provide remap functions of up to eight words in the program code in the CODE memory region. This enables applications stored on a non-erasable, ROM-based microcontroller to be patched if a small programmable memory, for example flash, is available in the device. During initialization, the application in ROM detects, from the programmable memory, whether a patch is required. If a patch is required, the application programs the FPB to remap a number of addresses. When those addresses are accessed, the accesses are redirected to a remap table specified in the FPB configuration, which means the program in the non-modifiable ROM can be patched.

1.1.3 Cortex-M4 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 FPU
- code-patch ability for ROM system updates
- 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 Memory Protection Unit (MPU) for safety-critical applications
• extensive implementation-defined 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-M4 core peripherals

These 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.

Memory Protection Unit

The Memory Protection Unit (MPU) improves system reliability by defining the memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region.

Floating-point Unit

The Floating-Point Unit (FPU) provides IEEE754-compliant operations on single-precision, 32-bit, floating-point values.
Chapter 2
The Cortex-M4 Processor

This chapter describes the Cortex-M4 processor. It contains the following sections:

- Programmers model on page 2-2
- Memory model on page 2-12
- Exception model on page 2-21
- Fault handling on page 2-29
- Power management on page 2-32.
2.1 Programmers model

This section describes the Cortex-M4 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>
<thead>
<tr>
<th>Processor mode</th>
<th>Used to execute</th>
<th>Privilege level for software execution</th>
<th>Stack used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Applications</td>
<td>Privileged or unprivileged&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Main stack or process stack&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>Handler</td>
<td>Exception handlers</td>
<td>Always privileged</td>
<td>Main stack</td>
</tr>
</tbody>
</table>

<sup>a</sup> See *CONTROL register on page 2-9*. 
2.1.3 Core registers

The processor core registers are:

- **Low registers**
  - SP (R13)
  - LR (R14)
  - PC (R15)

- **General-purpose registers**
  - R0
  - R1
  - R2
  - R3
  - R4
  - R5
  - R6
  - R7
  - R8
  - R9
  - R10
  - R11
  - R12

- **High registers**
  - MSP
  - PSP

- **Program status register**
  - PSR
  - PRIMASK
  - FAULTMASK
  - BASEPRI
  - CONTROL

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Required privilege</th>
<th>Reset value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0-R12</td>
<td>RW</td>
<td>Either</td>
<td>Unknown</td>
<td>General-purpose registers on page 2-4</td>
</tr>
<tr>
<td>MSP</td>
<td>RW</td>
<td>Privileged</td>
<td>See description</td>
<td>Stack Pointer on page 2-4</td>
</tr>
<tr>
<td>PSP</td>
<td>RW</td>
<td>Either</td>
<td>Unknown</td>
<td>Stack Pointer on page 2-4</td>
</tr>
<tr>
<td>LR</td>
<td>RW</td>
<td>Either</td>
<td>0xFFFFFFFF</td>
<td>Link Register on page 2-4</td>
</tr>
<tr>
<td>PC</td>
<td>RW</td>
<td>Either</td>
<td>See description</td>
<td>Program Counter on page 2-4</td>
</tr>
<tr>
<td>PSR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x01000000</td>
<td>Program Status Register on page 2-4</td>
</tr>
<tr>
<td>ASPR</td>
<td>RW</td>
<td>Either</td>
<td>Unknown</td>
<td>Application Program Status Register on page 2-5</td>
</tr>
<tr>
<td>IPSR</td>
<td>RO</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Program Status Register on page 2-6</td>
</tr>
<tr>
<td>EPSR</td>
<td>RO</td>
<td>Privileged</td>
<td>0x01000000</td>
<td>Execution Program Status Register on page 2-6</td>
</tr>
<tr>
<td>PRIMASK</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Priority Mask Register on page 2-8</td>
</tr>
<tr>
<td>FAULTMASK</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Fault Mask Register on page 2-8</td>
</tr>
<tr>
<td>BASEPRI</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Base Priority Mask Register on page 2-9</td>
</tr>
<tr>
<td>CONTROL</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>CONTROL register on page 2-9</td>
</tr>
</tbody>
</table>

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.
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 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, which is at 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 bitfields in the 32-bit PSR. The bit assignments are:

<table>
<thead>
<tr>
<th>31 30 29 28 27 26 25 24 23</th>
<th>16 15</th>
<th>10 9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>APSR</td>
<td>N</td>
<td>Z</td>
<td>C</td>
</tr>
<tr>
<td>IPSR</td>
<td>Reserved</td>
<td>ISR_NUMBER</td>
<td></td>
</tr>
<tr>
<td>EPSR</td>
<td>Reserved</td>
<td>ICI/IT</td>
<td>T</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Type</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSR</td>
<td>RW(^{a,b})</td>
<td>APSR, EPSR, and IPSR</td>
</tr>
<tr>
<td>IEPSR</td>
<td>RO</td>
<td>EPSR and IPSR</td>
</tr>
<tr>
<td>IAPSR</td>
<td>RW(^{a})</td>
<td>APSR and IPSR</td>
</tr>
<tr>
<td>EAPSR</td>
<td>RW(^{b})</td>
<td>APSR and EPSR</td>
</tr>
</tbody>
</table>

\(^{a}\) The processor ignores writes to the IPSR bits.
\(^{b}\) Reads of the EPSR bits return zero, and the processor ignores writes to these bits.

See the instruction descriptions *MRS on page 3-163* and *MSR on page 3-164* 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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>N</td>
<td>Negative flag</td>
</tr>
<tr>
<td>[30]</td>
<td>Z</td>
<td>Zero flag</td>
</tr>
<tr>
<td>[29]</td>
<td>C</td>
<td>Carry or borrow flag</td>
</tr>
<tr>
<td>[28]</td>
<td>V</td>
<td>Overflow flag</td>
</tr>
<tr>
<td>[27]</td>
<td>Q</td>
<td>DSP overflow and saturation flag</td>
</tr>
<tr>
<td>[26:20]</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>[15:0]</td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:9]</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>[8:0]</td>
<td>ISR_NUMBER</td>
<td>This is the number of the current exception:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Thread mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 = NMI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 = HardFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 = MemManage</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 = BusFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 = UsageFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7-10 = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = SVCall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 = Reserved for Debug</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14 = PendSV</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 = SysTick</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 = IRQ0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n+15 = IRQ(n-1)*</td>
</tr>
</tbody>
</table>

See Exception types on page 2-21 for more information.

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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:27]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[26:25], [15:10]</td>
<td>IT</td>
<td>Indicates the execution state bits of the IT instruction, see IT on page 3-122.</td>
</tr>
</tbody>
</table>
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 3-122 for more information.

**Thumb state**

The Cortex-M4 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 2-31 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 MSR instructions, or the CPS instruction to change the value of PRIMASK or FAULTMASK. See MRS on page 3-163, MSR on page 3-164, and CPS on page 3-159 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:

![PRIMASK register](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:1]</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>[0]</td>
<td>PRIMASK</td>
<td>0 = no effect 1 = prevents the activation of all exceptions with configurable priority.</td>
</tr>
</tbody>
</table>

**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:

![FAULTMASK register](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:1]</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>[0]</td>
<td>FAULTMASK</td>
<td>0 = no effect 1 = prevents the activation of all exceptions except for NMI.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
| [7:0]  | BASEPRI | 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. Register priority value fields are eight bits wide, and non-implemented low-order bits read as zero and ignore writes. 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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:3]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
| [2]   | FPCA | When floating-point is implemented this bit indicates whether context floating-point is currently active:  
|       |      | 0 = no floating-point context active  
|       |      | 1 = floating-point context active.  
|       |      | The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception. |
| [1]   | SPSEL | Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return:  
|       |      | 0 = MSP is the current stack pointer  
|       |      | 1 = PSP is the current stack pointer.  
| [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 2-17 on page 2-28.
In an OS environment, ARM recommends that threads running in Thread mode use the process stack and the kernel and exception handlers use the main stack.

By default, Thread mode uses the MSP. To switch the stack pointer used in Thread mode to the PSP, either:

- use the MSR instruction to set the Active stack pointer bit to 1, see MSR on page 3-164.
- perform an exception return to Thread mode with the appropriate EXC_RETURN value, see Table 2-17 on page 2-28.

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 3-162.

2.1.4 Exceptions and interrupts

The Cortex-M4 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 2-26 and Exception return on page 2-28 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
- manages all data memory accesses as little-endian or big-endian. Instruction memory and Private Peripheral Bus (PPB) accesses are always performed as little-endian. 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-M4 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-M4 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:

- *Power management programming hints on page 2-34*
- *CMSIS functions on page 3-9*
- *Accessing the Cortex-M4 NVIC registers using CMSIS on page 4-4*
- *NVIC programming hints on page 4-9.*
2.2 Memory model

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

```
+-----------------+-------------------+-------------------+
| Vendor-specific | Private peripheral | External device   |
| memory          | bus               | 1.0GB             |
+-----------------+-------------------+-------------------+
| 0xE01000000     | 0x0E0000000       | 0xA0000000        |
| 0xE0000000       | 0x00000000        | 0x9FFFFFFF        |
+-----------------+-------------------+-------------------+
| External RAM    | 1.0GB             |                  |
| 0xE20000000     | 0x00000000        |
| 0xE20000000     | 0x00000000        |
+-----------------+-------------------+-------------------+
| Peripheral      | 0.5GB             |
| 0x040000000     | 0x00000000        |
| 0x040000000     | 0x00000000        |
+-----------------+-------------------+-------------------+
| SRAM            | 0.5GB             |
| 0x20000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| Code            | 0.5GB             |
| 0x400000000     | 0x00000000        |
+-----------------+-------------------+-------------------+
| 32MB Bit band alias |                  |
| 0x20000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| Bit band region |                  |
| 0x22000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| Bit band alias  |                  |
| 0x42000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| Bit band region |                  |
| 0x43000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| 32MB Bit band alias |              |
| 0x40000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| 1MB Bit band region |                |
| 0x40000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
| 1MB Bit band alias |              |
| 0x42000000      | 0x00000000        |
+-----------------+-------------------+-------------------+
```

The regions for SRAM and peripherals include optional bit-band regions. Bit-banding provides atomic operations to bit data, see Optional bit-banding on page 2-16.

The processor reserves regions of the Private Peripheral Bus (PPB) address range for core peripheral registers, see About the Cortex-M4 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

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

**Strongly-ordered**

The processor preserves transaction order relative to all other transactions.

The different ordering requirements for Device and Strongly-ordered memory mean that the 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.

**Note**

This attribute is relevant only if the device is likely to be used in systems where memory is shared between multiple processors.

**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:

<table>
<thead>
<tr>
<th></th>
<th>Normal access</th>
<th>Device access</th>
<th>Strongly-ordered access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Normal access</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Device access, non-shareable</td>
<td>-</td>
<td>&lt;</td>
<td>-</td>
</tr>
<tr>
<td>Device access, shareable</td>
<td>-</td>
<td>-</td>
<td>&lt;</td>
</tr>
<tr>
<td>Strongly-ordered access</td>
<td>-</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Address range</th>
<th>Memory region</th>
<th>Memory type</th>
<th>XN</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000-0x1FFFFFF</td>
<td>Code</td>
<td>Normal</td>
<td>-</td>
<td>Executable region for program code. You can also put data here.</td>
</tr>
<tr>
<td>0x20000000-0x3FFFFFF</td>
<td>SRAM</td>
<td>Normal</td>
<td>-</td>
<td>Executable region for data. You can also put code here. This region includes bit band and bit band alias areas, see Table 2-13 on page 2-16.</td>
</tr>
<tr>
<td>0x40000000-0x5FFFFFF</td>
<td>Peripheral</td>
<td>Device</td>
<td>XN</td>
<td>This region includes bit band and bit band alias areas, see Table 2-14 on page 2-16.</td>
</tr>
<tr>
<td>0x60000000-0x9FFFFFF</td>
<td>External RAM</td>
<td>Normal</td>
<td>-</td>
<td>Executable region for data.</td>
</tr>
<tr>
<td>0xA0000000-0xDFFFFFF</td>
<td>External device</td>
<td>Device</td>
<td>XN</td>
<td>External Device memory.</td>
</tr>
<tr>
<td>0xE0000000-0xE0000000</td>
<td>Private Peripheral Bus</td>
<td>Strongly-ordered</td>
<td>XN</td>
<td>This region includes the NVIC, System timer, and system control block.</td>
</tr>
<tr>
<td>0xE0100000-0xFFFFFFFF</td>
<td>Device</td>
<td>Device</td>
<td>XN</td>
<td>Implementation-specific.</td>
</tr>
</tbody>
</table>

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

The Code, SRAM, and external RAM regions can hold programs. However, ARM recommends that programs always use the Code region. This is because the processor has separate buses that enable instruction fetches and data accesses to occur simultaneously.

The optional MPU can override the default memory access behavior described in this section. For more information, see Optional Memory Protection Unit on page 4-37.

### Additional memory access constraints for caches and shared memory

When a system includes caches or shared memory, some memory regions might have additional access constraints, and some regions are subdivided, as Table 2-12 shows:

### Table 2-12 Memory region shareability and cache policies

<table>
<thead>
<tr>
<th>Address range</th>
<th>Memory region</th>
<th>Memory type</th>
<th>Shareability</th>
<th>Cache policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000-0x1FFFFFF</td>
<td>Code</td>
<td>Normal</td>
<td>-</td>
<td>WT</td>
</tr>
<tr>
<td>0x20000000-0x3FFFFFF</td>
<td>SRAM</td>
<td>Normal</td>
<td>-</td>
<td>WBWA</td>
</tr>
<tr>
<td>0x40000000-0x5FFFFFF</td>
<td>Peripheral</td>
<td>Device</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0x60000000-0x7FFFFFF</td>
<td>External RAM</td>
<td>Normal</td>
<td>-</td>
<td>WBWA</td>
</tr>
<tr>
<td>0x80000000-0x9FFFFFF</td>
<td></td>
<td></td>
<td></td>
<td>WT</td>
</tr>
</tbody>
</table>

The Code, SRAM, and external RAM regions can hold programs. However, ARM recommends that programs always use the Code region. This is because the processor has separate buses that enable instruction fetches and data accesses to occur simultaneously.

The optional MPU can override the default memory access behavior described in this section. For more information, see Optional Memory Protection Unit on page 4-37.

### Additional memory access constraints for caches and shared memory

When a system includes caches or shared memory, some memory regions might have additional access constraints, and some regions are subdivided, as Table 2-12 shows:

### Table 2-12 Memory region shareability and cache policies

<table>
<thead>
<tr>
<th>Address range</th>
<th>Memory region</th>
<th>Memory type</th>
<th>Shareability</th>
<th>Cache policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000-0x1FFFFFF</td>
<td>Code</td>
<td>Normal</td>
<td>-</td>
<td>WT</td>
</tr>
<tr>
<td>0x20000000-0x3FFFFFF</td>
<td>SRAM</td>
<td>Normal</td>
<td>-</td>
<td>WBWA</td>
</tr>
<tr>
<td>0x40000000-0x5FFFFFF</td>
<td>Peripheral</td>
<td>Device</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0x60000000-0x7FFFFFF</td>
<td>External RAM</td>
<td>Normal</td>
<td>-</td>
<td>WBWA</td>
</tr>
<tr>
<td>0x80000000-0x9FFFFFF</td>
<td></td>
<td></td>
<td></td>
<td>WT</td>
</tr>
</tbody>
</table>
Instruction prefetch and branch prediction

The Cortex-M4 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 3-160.

DSB

The Data Synchronization Barrier (DSB) instruction ensures that outstanding memory transactions complete before subsequent instructions execute. See DSB on page 3-161.

ISB

The Instruction Synchronization Barrier (ISB) ensures that the effect of all completed memory transactions is recognizable by subsequent instructions. See ISB on page 3-162.

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.
2.2.5 Optional bit-banding

A bit-band region maps each word in a bit-band alias region to a single bit in the bit-band region. The bit-band regions occupy the lowest 1MB of the SRAM and peripheral memory regions.

The memory map has two 32MB alias regions that map to two 1MB bit-band regions:

- accesses to the 32MB SRAM alias region map to the 1MB SRAM bit-band region, as shown in Table 2-13
- accesses to the 32MB peripheral alias region map to the 1MB peripheral bit-band region, as shown in Table 2-14.

### Table 2-13 SRAM memory bit-banding regions

<table>
<thead>
<tr>
<th>Address range</th>
<th>Memory region</th>
<th>Instruction and data accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20000000-0x20000000</td>
<td>SRAM bit-band region</td>
<td>Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.</td>
</tr>
<tr>
<td>0x22000000-0x23000000</td>
<td>SRAM bit-band alias</td>
<td>Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not remapped.</td>
</tr>
</tbody>
</table>

### Table 2-14 Peripheral memory bit-banding regions

<table>
<thead>
<tr>
<th>Address range</th>
<th>Memory region</th>
<th>Instruction and data accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40000000-0x40000000</td>
<td>Peripheral bit-band alias</td>
<td>Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.</td>
</tr>
<tr>
<td>0x42000000-0x43000000</td>
<td>Peripheral bit-band region</td>
<td>Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not permitted.</td>
</tr>
</tbody>
</table>

#### Note

- A word access to the SRAM or peripheral bit-band alias regions maps to a single bit in the SRAM or peripheral bit-band region
- Bit band accesses can use byte, halfword, or word transfers. The bit band transfer size matches the transfer size of the instruction making the bit band access.

The following formula shows how the alias region maps onto the bit-band region:

$$bit\_word\_offset = (byte\_offset \times 32) + (bit\_number \times 4)$$

$$bit\_word\_addr = bit\_band\_base + bit\_word\_offset$$

where:

- **Bit_word_offset** is the position of the target bit in the bit-band memory region
- **Bit_word_addr** is the address of the word in the alias memory region that maps to the targeted bit
- **Bit_band_base** is the starting address of the alias region
- **Byte_offset** is the number of the byte in the bit-band region that contains the targeted bit
Bit_number is the bit position, 0-7, of the targeted bit.

Figure 2-1 shows examples of bit-band mapping between the SRAM bit-band alias region and the SRAM bit-band region:

- the alias word at 0x23FFFFE0 maps to bit[0] of the bit-band byte at 0x200FFFFF: 
  \[0x23FFFFE0 = 0x22000000 + (0xFFFFF\times32) + (0\times4)\]

- the alias word at 0x23FFFFFC maps to bit[7] of the bit-band byte at 0x200FFFFF: 
  \[0x23FFFFFC = 0x22000000 + (0xFFFFF\times32) + (7\times4)\]

- the alias word at 0x22000000 maps to bit[0] of the bit-band byte at 0x20000000: 
  \[0x22000000 = 0x22000000 + (0\times32) + (0\times4)\]

- the alias word at 0x2200001C maps to bit[7] of the bit-band byte at 0x20000000: 
  \[0x2200001C = 0x22000000 + (0\times32) + (7\times4)\].

![32MB alias region](image)

**Directly accessing an alias region**

Writing to a word in the alias region updates a single bit in the bit-band region. Bit[0] of the value written to a word in the alias region determines the value written to the targeted bit in the bit-band region. Writing a value with bit[0] set to 1 writes a 1 to the bit-band bit, and writing a value with bit[0] set to 0 writes a 0 to the bit-band bit.

Bits[31:1] of the alias word have no effect on the bit-band bit. Writing 0x01 has the same effect as writing 0xFF. Writing 0x00 has the same effect as writing 0x0E.

Reading a word in the alias region:

- 0x00000000 indicates that the targeted bit in the bit-band region is set to zero
- 0x00000001 indicates that the targeted bit in the bit-band region is set to 1

**Directly accessing a bit-band region**

*Behavior of memory accesses on page 2-14* describes the behavior of direct byte, halfword, or word accesses to the bit-band regions.
2.2.6 Memory endianness

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example, bytes 0-3 hold the first stored word, and bytes 4-7 hold the second stored word. The memory endianness used is implementation-defined, and the following subsections describe the possible implementations:

- **Byte-invariant big-endian format**
- **Little-endian format**

Read the AICR.ENDIANNESS field to find the implemented endianness, see Application Interrupt and Reset Control Register on page 4-16.

### 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
A   B0 | msbyte
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
A   B0 | lsbyte
A+1 B1
A+2 B2
A+3 B3 | msbyte
```

2.2.7 Synchronization primitives

The Cortex-M4 instruction set 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 semaphores 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-M4 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, 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 Store-Exclusive 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 Store-Exclusive instruction, or an exception, removes the local exclusive access tags, and global exclusive access tags for the processor.

For more information about the synchronization primitive instructions, see LDREX and STREX on page 3-36 and CLREX on page 3-38.

### 2.2.8 Programming hints for the synchronization primitives

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>CMSIS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDREX</td>
<td>uint32_t __LDREXW (uint32_t *addr)</td>
</tr>
<tr>
<td>LDREXH</td>
<td>uint16_t __LDREXH (uint16_t *addr)</td>
</tr>
<tr>
<td>LDREXB</td>
<td>uint8_t __LDREXB (uint8_t *addr)</td>
</tr>
<tr>
<td>STREX</td>
<td>uint32_t __STREXW (uint32_t value, uint32_t *addr)</td>
</tr>
<tr>
<td>STREXH</td>
<td>uint32_t __STREXH (uint16_t value, uint16_t *addr)</td>
</tr>
<tr>
<td>STREXB</td>
<td>uint32_t __STREXB (uint8_t value, uint8_t *addr)</td>
</tr>
<tr>
<td>CLREX</td>
<td>void __CLREX (void)</td>
</tr>
</tbody>
</table>
2.3 Exception model

This section describes the exception model. It describes:
- Exception states
- Exception types
- Exception handlers on page 2-23
- Vector table on page 2-23
- Exception priorities on page 2-24
- Interrupt priority grouping on page 2-25
- Exception entry and return on page 2-25.

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. 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 signalled 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. HardFaualts 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 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 signalled 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 2-16 Properties of the different exception types**

<table>
<thead>
<tr>
<th>Exception numbera</th>
<th>IRQ numbera</th>
<th>Exception type</th>
<th>Priority</th>
<th>Vector address or offsetb</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>Reset</td>
<td>-3, the highest</td>
<td>0x00000004</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>2</td>
<td>-14</td>
<td>NMI</td>
<td>-2</td>
<td>0x00000008</td>
<td>Asynchronous</td>
</tr>
<tr>
<td>3</td>
<td>-13</td>
<td>HardFault</td>
<td>-1</td>
<td>0x0000000C</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-12</td>
<td>MemManage</td>
<td>Configurablec</td>
<td>0x00000010</td>
<td>Synchronous</td>
</tr>
<tr>
<td>5</td>
<td>-11</td>
<td>BusFault</td>
<td>Configurablec</td>
<td>0x00000014</td>
<td>Synchronous when precise, asynchronous when imprecise</td>
</tr>
<tr>
<td>6</td>
<td>-10</td>
<td>UsageFault</td>
<td>Configurablec</td>
<td>0x00000018</td>
<td>Synchronous</td>
</tr>
<tr>
<td>7-10</td>
<td>-</td>
<td>Reserved</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
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 2-16 on page 2-22 shows as having configurable priority, see:

- System Handler Control and State Register on page 4-23
- Interrupt Clear-enable Registers on page 4-5.

For more information about HardFaults, MemManage faults, BusFaults, and UsageFaults, see Fault handling on page 2-29.

### 2.3.3 Exception handlers

The processor handles exceptions using:

**Interrupt Service Routines (ISRs)**

The IRQ interrupts are the exceptions handled by ISRs.

**Fault handlers**

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

**System handlers**

NMI, PendSV, SVC, and 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 2-2 on page 2-24 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.
On system reset, the vector table is fixed at address 0x00000000. Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range 0x00000080 to 0x3FFFFF80, see Vector Table Offset Register on page 4-16.

### 2.3.5 Exception priorities

As Table 2-16 on page 2-22 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 4-21
- Interrupt Priority Registers on page 4-7.

_____ Note  _____

Configurable priority values are in the range 0-. 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 asserted 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 4-16.

### 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 on page 2-26 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 2-28 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-M4 processor automatically stacks the architected floating-point state on exception entry. Figure 2-3 on page 2-27 shows the Cortex-M4 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 2-3 on page 2-27 shows this stack frame also.
Immediately after stacking, the stack pointer indicates the lowest address in the stack frame. The alignment of the stack frame is controlled via the STKALIGN bit of the Configuration Control Register (CCR).

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 2-17 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 2-17 Exception return behavior

<table>
<thead>
<tr>
<th>EXC_RETURN[31:0]</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFF1</td>
<td>Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.</td>
</tr>
<tr>
<td>0xFFFFFFF9</td>
<td>Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.</td>
</tr>
<tr>
<td>0xFFFFFFD</td>
<td>Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.</td>
</tr>
<tr>
<td>0xFFFFFE1</td>
<td>Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.</td>
</tr>
<tr>
<td>0xFFFFFE9</td>
<td>Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.</td>
</tr>
<tr>
<td>0xFFFFFED</td>
<td>Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.</td>
</tr>
</tbody>
</table>
2.4 Fault handling

Faults are a subset of the exceptions, see Exception model on page 2-21. 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 2-18 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 Configurable Fault Status Register on page 4-24 for more information about the fault status registers.

<table>
<thead>
<tr>
<th>Fault</th>
<th>Handler</th>
<th>Bit name</th>
<th>Fault status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus error on a vector read</td>
<td>HardFault</td>
<td>VECTTBL</td>
<td>HardFault Status Register on page 4-30</td>
</tr>
<tr>
<td>Fault escalated to a hard fault</td>
<td></td>
<td>FORCED</td>
<td></td>
</tr>
<tr>
<td>MPU or default memory map mismatch: on instruction access</td>
<td>MemManage</td>
<td>-</td>
<td>MemManage Fault Address Register on page 4-30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IACCVIOL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DACCVIOL</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSTKERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUNSKERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MLSPERR</td>
<td></td>
</tr>
<tr>
<td>Bus error: on data access</td>
<td>BusFault</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>STKERR</td>
<td>BusFault Status Register on page 4-26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>UNSTKERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IBUSERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LSPERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PRECISERR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMPRECISERR</td>
<td></td>
</tr>
</tbody>
</table>
### 2.4.2 Fault escalation and hard faults

All faults exceptions except for HardFault have configurable exception priority, see *System Handler Priority Registers* on page 4-21. Software can disable execution of the handlers for these faults, see *System Handler Control and State Register* on page 4-23.

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 2-21.

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 Fault status registers and fault address registers

The fault status registers indicate the cause of a fault. For BusFaults and MemManage faults, the fault address register indicates the address accessed by the operation that caused the fault, as shown in Table 2-19.

<table>
<thead>
<tr>
<th>Handler</th>
<th>Status register name</th>
<th>Address register name</th>
<th>Register description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HardFault</td>
<td>HFSR</td>
<td>-</td>
<td>HardFault Status Register on page 4-30</td>
</tr>
<tr>
<td>MemManage</td>
<td>MMFSR</td>
<td>MMFAR</td>
<td>MemManage Fault Status Register on page 4-25</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MemManage Fault Address Register on page 4-30</td>
</tr>
<tr>
<td>BusFault</td>
<td>BFSR</td>
<td>BFAR</td>
<td>BusFault Status Register on page 4-26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>BusFault Address Register on page 4-31</td>
</tr>
<tr>
<td>UsageFault</td>
<td>UFSR</td>
<td>-</td>
<td>UsageFault Status Register on page 4-28</td>
</tr>
</tbody>
</table>

2.4.4 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-M4 processor sleep modes reduce power consumption. The sleep modes your device implements are implementation-defined. The modes can be 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 of the SCR selects which sleep mode is used, see System Control Register on page 4-19. 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, \texttt{WFI}, causes immediate entry to sleep mode unless the wake-up condition is true, see Wakeup from WFI or sleep-on-exit on page 2-33. When the processor executes a \texttt{WFI} instruction it stops executing instructions and enters sleep mode. See \texttt{WFI} on page 3-169 for more information.

**Wait for event**

The *Wait For Event* instruction, \texttt{WFE}, causes entry to sleep mode depending on the value of a one-bit event register. When the processor executes a \texttt{WFE} instruction, it checks the value of the event register:

- \texttt{0} The processor stops executing instructions and enters sleep mode.
- \texttt{1} The processor clears the register to \texttt{0} and continues executing instructions without entering sleep mode.

See \texttt{WFE} on page 3-168 for more information.

If the event register is \texttt{1}, this indicates that the processor must not enter sleep mode on execution of a \texttt{WFE} instruction. Typically, this is because an external event signal is asserted, or a processor in the system has executed an \texttt{SEV} instruction, see \texttt{SEV} on page 3-166. Software cannot access this register directly.

**Sleep-on-exit**

If the SLEEPONEXIT bit of the SCR is set to \texttt{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 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, see The 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 4-19.

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 4-19.

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-M4 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 external event input

Your device might include an external event input signal, so that device peripherals can signal the processor, to either:

• wake the processor from WFE
• 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 2-32* and the documentation supplied by your device vendor for more information about this signal.

### 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:

```c
void __WFE(void) // Wait for Event
void __WFI(void) // Wait for Interrupt
```
Chapter 3
The Cortex-M4 Instruction Set

This chapter is the reference material for the Cortex-M4 instruction set description in a User Guide. The following sections give general information:

- *Instruction set summary* on page 3-2
- *CMSIS functions* on page 3-9
- *About the instruction descriptions* on page 3-11.

Each of the following sections describes a functional group of Cortex-M4 instructions. Together they describe all the instructions supported by the Cortex-M4 processor:

- *Memory access instructions* on page 3-22
- *General data processing instructions* on page 3-39
- *Multiply and divide instructions* on page 3-74
- *Saturating instructions* on page 3-95
- *Packing and unpacking instructions* on page 3-107
- *Bitfield instructions* on page 3-114
- *Branch and control instructions* on page 3-118
- *Miscellaneous instructions* on page 3-157
- *Floating-point instructions* on page 3-126.
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-M4 instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC, ADCS</td>
<td>(Rd,) Rn, Op2</td>
<td>Add with Carry</td>
<td>N,Z,C,V</td>
<td>page 3-41</td>
</tr>
<tr>
<td>ADD, ADDS</td>
<td>(Rd,) Rn, Op2</td>
<td>Add</td>
<td>N,Z,C,V</td>
<td>page 3-41</td>
</tr>
<tr>
<td>ADD, ADDW</td>
<td>(Rd,) Rn, #imm12</td>
<td>Add</td>
<td>-</td>
<td>page 3-41</td>
</tr>
<tr>
<td>ADR</td>
<td>Rd, label</td>
<td>Load PC-relative Address</td>
<td>-</td>
<td>page 3-23</td>
</tr>
<tr>
<td>AND, ANDS</td>
<td>(Rd,) Rn, Op2</td>
<td>Logical AND</td>
<td>N,Z,C</td>
<td>page 3-44</td>
</tr>
<tr>
<td>ASR, ASRS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Arithmetic Shift Right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>B</td>
<td>label</td>
<td>Branch</td>
<td>-</td>
<td>page 3-119</td>
</tr>
<tr>
<td>BFC</td>
<td>Rd, #lsb, #width</td>
<td>Bit Field Clear</td>
<td>-</td>
<td>page 3-115</td>
</tr>
<tr>
<td>BFI</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Bit Field Insert</td>
<td>-</td>
<td>page 3-115</td>
</tr>
<tr>
<td>BIC, BICS</td>
<td>(Rd,) Rn, Op2</td>
<td>Bit Clear</td>
<td>N,Z,C</td>
<td>page 3-44</td>
</tr>
<tr>
<td>BKPT</td>
<td>#imm</td>
<td>Breakpoint</td>
<td>-</td>
<td>page 3-158</td>
</tr>
<tr>
<td>BL</td>
<td>label</td>
<td>Branch with Link</td>
<td>-</td>
<td>page 3-119</td>
</tr>
<tr>
<td>BLX</td>
<td>Rm</td>
<td>Branch indirect with Link</td>
<td>-</td>
<td>page 3-119</td>
</tr>
<tr>
<td>BX</td>
<td>Rm</td>
<td>Branch indirect</td>
<td>-</td>
<td>page 3-119</td>
</tr>
<tr>
<td>CBNZ</td>
<td>Rn, label</td>
<td>Compare and Branch if Non Zero</td>
<td>-</td>
<td>page 3-121</td>
</tr>
<tr>
<td>CBZ</td>
<td>Rn, label</td>
<td>Compare and Branch if Zero</td>
<td>-</td>
<td>page 3-121</td>
</tr>
<tr>
<td>CLREX</td>
<td>-</td>
<td>Clear Exclusive</td>
<td>-</td>
<td>page 3-38</td>
</tr>
<tr>
<td>CLZ</td>
<td>Rd, Rm</td>
<td>Count Leading Zeros</td>
<td>-</td>
<td>page 3-48</td>
</tr>
<tr>
<td>CMN</td>
<td>Rn, Op2</td>
<td>Compare Negative</td>
<td>N,Z,C,V</td>
<td>page 3-49</td>
</tr>
<tr>
<td>CMP</td>
<td>Rn, Op2</td>
<td>Compare</td>
<td>N,Z,C,V</td>
<td>page 3-49</td>
</tr>
<tr>
<td>CPSID</td>
<td>i</td>
<td>Change Processor State, Disable Interrupts</td>
<td>-</td>
<td>page 3-159</td>
</tr>
<tr>
<td>CPSIE</td>
<td>i</td>
<td>Change Processor State, Enable Interrupts</td>
<td>-</td>
<td>page 3-159</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Brief description</td>
<td>Flags</td>
<td>Page</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------</td>
<td>----------------------------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>DMB</td>
<td></td>
<td>Data Memory Barrier</td>
<td>-</td>
<td>page 3-160</td>
</tr>
<tr>
<td>DSB</td>
<td></td>
<td>Data Synchronization Barrier</td>
<td>-</td>
<td>page 3-161</td>
</tr>
<tr>
<td>EOR, EORS</td>
<td>(Rd,) Rn, Op2</td>
<td>Exclusive OR</td>
<td>N,Z,C</td>
<td>page 3-44</td>
</tr>
<tr>
<td>ISB</td>
<td></td>
<td>Instruction Synchronization Barrier</td>
<td>-</td>
<td>page 3-162</td>
</tr>
<tr>
<td>IT</td>
<td></td>
<td>If-Then condition block</td>
<td>-</td>
<td>page 3-122</td>
</tr>
<tr>
<td>LDM</td>
<td>Rn(), reglist</td>
<td>Load Multiple registers, increment after</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>LDMDB, LDMEA</td>
<td>Rn(), reglist</td>
<td>Load Multiple registers, decrement before</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>LDMFD, LDMIA</td>
<td>Rn(), reglist</td>
<td>Load Multiple registers, increment after</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>LDR</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with word</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LDRB, LDRBT</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with byte</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LDRD</td>
<td>Rt, Rt2, [Rn, #offset]</td>
<td>Load Register with two bytes</td>
<td>-</td>
<td>page 3-24</td>
</tr>
<tr>
<td>LDREX</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register Exclusive</td>
<td>-</td>
<td>page 3-36</td>
</tr>
<tr>
<td>LDREXH</td>
<td>Rt, [Rn]</td>
<td>Load Register Exclusive with Halfword</td>
<td>-</td>
<td>page 3-36</td>
</tr>
<tr>
<td>LDRH, LDRHT</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with Halfword</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LDRSB, LDRSBT</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with Signed Byte</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LDRSH, LDRSHT</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with Signed Halfword</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LDRT</td>
<td>Rt, [Rn, #offset]</td>
<td>Load Register with word</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>LSL, LSLS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Logical Shift Left</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>LSR, LSRS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Logical Shift Right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>MLA</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Multiply with Accumulate, 32-bit result</td>
<td>-</td>
<td>page 3-75</td>
</tr>
<tr>
<td>MLS</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Multiply and Subtract, 32-bit result</td>
<td>-</td>
<td>page 3-75</td>
</tr>
<tr>
<td>MOV, MOVS</td>
<td>Rd, Op2</td>
<td>Move</td>
<td>N,Z,C</td>
<td>page 3-50</td>
</tr>
<tr>
<td>MOVT</td>
<td>Rd, #imm16</td>
<td>Move Top</td>
<td>-</td>
<td>page 3-52</td>
</tr>
<tr>
<td>MOVW, MOV</td>
<td>Rd, #imm16</td>
<td>Move 16-bit constant</td>
<td>N,Z,C</td>
<td>page 3-50</td>
</tr>
<tr>
<td>MRS</td>
<td>Rd, spec_reg</td>
<td>Move from Special Register to general register</td>
<td>-</td>
<td>page 3-163</td>
</tr>
<tr>
<td>MSR</td>
<td>spec_reg, Rm</td>
<td>Move from general register to Special Register</td>
<td>N,Z,C,V</td>
<td>page 3-164</td>
</tr>
<tr>
<td>MUL, MULS</td>
<td>(Rd,) Rn, Rm</td>
<td>Multiply, 32-bit result</td>
<td>N,Z</td>
<td>page 3-75</td>
</tr>
<tr>
<td>MVN, MVNS</td>
<td>Rd, Op2</td>
<td>Move NOT</td>
<td>N,Z,C</td>
<td>page 3-50</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>No Operation</td>
<td>-</td>
<td>page 3-165</td>
</tr>
<tr>
<td>ORN, ORNS</td>
<td>(Rd,) Rn, Op2</td>
<td>Logical OR NOT</td>
<td>N,Z,C</td>
<td>page 3-44</td>
</tr>
<tr>
<td>ORR, ORRS</td>
<td>(Rd,) Rn, Op2</td>
<td>Logical OR</td>
<td>N,Z,C</td>
<td>page 3-44</td>
</tr>
<tr>
<td>PKHTB, PKHBT</td>
<td>(Rd,) Rn, Op2</td>
<td>Pack Halfword</td>
<td>-</td>
<td>page 3-108</td>
</tr>
</tbody>
</table>
Table 3-1 Cortex-M4 instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP</td>
<td>reglist</td>
<td>Pop registers from stack</td>
<td>-</td>
<td>page 3-34</td>
</tr>
<tr>
<td>PUSH</td>
<td>reglist</td>
<td>Push registers onto stack</td>
<td>-</td>
<td>page 3-34</td>
</tr>
<tr>
<td>QADD</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating double and Add</td>
<td>Q</td>
<td>page 3-98</td>
</tr>
<tr>
<td>QADD16</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Add 16</td>
<td>-</td>
<td>page 3-98</td>
</tr>
<tr>
<td>QADD8</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Add 8</td>
<td>-</td>
<td>page 3-98</td>
</tr>
<tr>
<td>QASX</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Add and Subtract with Exchange</td>
<td>-</td>
<td>page 3-100</td>
</tr>
<tr>
<td>QDADD</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Add</td>
<td>Q</td>
<td>page 3-102</td>
</tr>
<tr>
<td>QDSUB</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating double and Subtract</td>
<td>Q</td>
<td>page 3-102</td>
</tr>
<tr>
<td>QSAX</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Subtract and Add with Exchange</td>
<td>-</td>
<td>page 3-100</td>
</tr>
<tr>
<td>QSUB</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Subtract</td>
<td>Q</td>
<td>page 3-102</td>
</tr>
<tr>
<td>QSUB16</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Subtract 16</td>
<td>-</td>
<td>page 3-102</td>
</tr>
<tr>
<td>QSUB8</td>
<td>(Rd,), Rn, Rm</td>
<td>Saturating Subtract 8</td>
<td>-</td>
<td>page 3-102</td>
</tr>
<tr>
<td>RBIT</td>
<td>Rd, Rn</td>
<td>Reverse Bits</td>
<td>-</td>
<td>page 3-53</td>
</tr>
<tr>
<td>REV</td>
<td>Rd, Rn</td>
<td>Reverse byte order in a word</td>
<td>-</td>
<td>page 3-53</td>
</tr>
<tr>
<td>REV16</td>
<td>Rd, Rn</td>
<td>Reverse byte order in each halfword</td>
<td>-</td>
<td>page 3-53</td>
</tr>
<tr>
<td>REVSH</td>
<td>Rd, Rn</td>
<td>Reverse byte order in bottom halfword and sign extend</td>
<td>-</td>
<td>page 3-53</td>
</tr>
<tr>
<td>ROR, RORS</td>
<td>Rd, Rm, &lt;Rs</td>
<td>#n&gt;</td>
<td>Rotate Right</td>
<td>N,Z,C</td>
</tr>
<tr>
<td>RRX, RRXS</td>
<td>Rd, Rm</td>
<td>Rotate Right with Extend</td>
<td>N,Z,C</td>
<td>page 3-46</td>
</tr>
<tr>
<td>RSB, RSBX</td>
<td>(Rd,), Rn, Op2</td>
<td>Reverse Subtract</td>
<td>N,Z,C,V</td>
<td>page 3-41</td>
</tr>
<tr>
<td>SADD16</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Add 16</td>
<td>GE</td>
<td>page 3-54</td>
</tr>
<tr>
<td>SADD8</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Add 8</td>
<td>GE</td>
<td>page 3-54</td>
</tr>
<tr>
<td>SASX</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Add and Subtract with Exchange</td>
<td>GE</td>
<td>page 3-60</td>
</tr>
<tr>
<td>SBC, SBCS</td>
<td>(Rd,), Rn, Op2</td>
<td>Subtract with Carry</td>
<td>N,Z,C,V</td>
<td>page 3-41</td>
</tr>
<tr>
<td>SBFX</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Signed Bit Field Extract</td>
<td>-</td>
<td>page 3-116</td>
</tr>
<tr>
<td>SDIV</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Divide</td>
<td>-</td>
<td>page 3-94</td>
</tr>
<tr>
<td>SEL</td>
<td>(Rd,), Rn, Rm</td>
<td>Select bytes</td>
<td>-</td>
<td>page 3-70</td>
</tr>
<tr>
<td>SEV</td>
<td>-</td>
<td>Send Event</td>
<td>-</td>
<td>page 3-166</td>
</tr>
<tr>
<td>SADD16</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Add 16</td>
<td>-</td>
<td>page 3-55</td>
</tr>
<tr>
<td>SADD8</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Add 8</td>
<td>-</td>
<td>page 3-55</td>
</tr>
<tr>
<td>SHASX</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Add and Subtract with Exchange</td>
<td>-</td>
<td>page 3-56</td>
</tr>
<tr>
<td>SHSAX</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Subtract and Add with Exchange</td>
<td>-</td>
<td>page 3-56</td>
</tr>
<tr>
<td>SHSUB16</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Subtract 16</td>
<td>-</td>
<td>page 3-58</td>
</tr>
<tr>
<td>SHSUB8</td>
<td>(Rd,), Rn, Rm</td>
<td>Signed Halving Subtract 8</td>
<td>-</td>
<td>page 3-58</td>
</tr>
</tbody>
</table>
### Table 3-1 Cortex-M4 instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLABB, SMLABT, SMLATB, SMLATT</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Multiply Accumulate Long (halfwords)</td>
<td>Q</td>
<td>page 3-79</td>
</tr>
<tr>
<td>SMLAD, SMLADX</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Multiply Accumulate Dual</td>
<td>Q</td>
<td>page 3-81</td>
</tr>
<tr>
<td>SMLAL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed Multiply with Accumulate (32 x 32 + 64), 64-bit result</td>
<td>-</td>
<td>page 3-93</td>
</tr>
<tr>
<td>SMLALBB, SMLALBT, SMLALTB, SMLALTT</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed Multiply Accumulate Long, halfwords</td>
<td>-</td>
<td>page 3-82</td>
</tr>
<tr>
<td>SMLAD, SMLADX</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed Multiply Accumulate Long Dual</td>
<td>-</td>
<td>page 3-82</td>
</tr>
<tr>
<td>SMLAWB, SMLAWT</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Multiply Accumulate, word by halfword</td>
<td>Q</td>
<td>page 3-79</td>
</tr>
<tr>
<td>SMLSD</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Multiply Subtract Dual</td>
<td>Q</td>
<td>page 3-84</td>
</tr>
<tr>
<td>SMLSD</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed Multiply Subtract Long Dual</td>
<td>-</td>
<td>page 3-84</td>
</tr>
<tr>
<td>SMLA</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Most significant word Multiply Accumulate</td>
<td>-</td>
<td>page 3-86</td>
</tr>
<tr>
<td>SMMLA</td>
<td>Rd, Rn, Rm, Ra</td>
<td>Signed Most significant word Multiply Subtract</td>
<td>-</td>
<td>page 3-86</td>
</tr>
<tr>
<td>SMMUL, SMMLUR</td>
<td>Rd, Rn, Rm</td>
<td>Signed Most significant word Multiply</td>
<td>-</td>
<td>page 3-88</td>
</tr>
<tr>
<td>SMUAD</td>
<td>Rd, Rn, Rm</td>
<td>Signed dual Multiply Add</td>
<td>Q</td>
<td>page 3-89</td>
</tr>
<tr>
<td>SMULL, SMULBT, SMULTB, SMULTT</td>
<td>Rd, Rn, Rm</td>
<td>Signed Multiply (halfwords)</td>
<td>-</td>
<td>page 3-91</td>
</tr>
<tr>
<td>SMULL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Signed Multiply (32 x 32), 64-bit result</td>
<td>-</td>
<td>page 3-93</td>
</tr>
<tr>
<td>SMULWB, SMULWT</td>
<td>Rd, Rn, Rm</td>
<td>Signed Multiply word by halfword</td>
<td>-</td>
<td>page 3-91</td>
</tr>
<tr>
<td>SMUSD, SMUSDX</td>
<td>Rd, Rn, Rm</td>
<td>Signed dual Multiply Subtract</td>
<td>-</td>
<td>page 3-89</td>
</tr>
<tr>
<td>SSAT</td>
<td>Rd, #n, Rm {,shift #s}</td>
<td>Signed Saturate</td>
<td>Q</td>
<td>page 3-96</td>
</tr>
<tr>
<td>SSAT16</td>
<td>Rd, #n, Rm</td>
<td>Signed Saturate 16</td>
<td>Q</td>
<td>page 3-97</td>
</tr>
<tr>
<td>SSAX</td>
<td>Rd, Rn, Rm</td>
<td>Signed Subtract and Add with Exchange</td>
<td>GE</td>
<td>page 3-60</td>
</tr>
<tr>
<td>SSUB16</td>
<td>Rd, Rn, Rm</td>
<td>Signed Subtract 16</td>
<td>-</td>
<td>page 3-59</td>
</tr>
<tr>
<td>SSUB8</td>
<td>Rd, Rn, Rm</td>
<td>Signed Subtract 8</td>
<td>-</td>
<td>page 3-59</td>
</tr>
<tr>
<td>STM</td>
<td>Rn(!), reglist</td>
<td>Store Multiple registers, increment after</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>STMD8, STMEA</td>
<td>Rn(!), reglist</td>
<td>Store Multiple registers, decrement before</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>STMFD, STMIA</td>
<td>Rn(!), reglist</td>
<td>Store Multiple registers, increment after</td>
<td>-</td>
<td>page 3-32</td>
</tr>
<tr>
<td>STR</td>
<td>Rt, [Rn, #offset]</td>
<td>Store Register word</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>STR8, STR8T</td>
<td>Rt, [Rn, #offset]</td>
<td>Store Register byte</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>STRD</td>
<td>Rt, Rt2, [Rn, #offset]</td>
<td>Store Register two words</td>
<td>-</td>
<td>page 3-24</td>
</tr>
<tr>
<td>STREX</td>
<td>Rd, Rt, [Rn, #offset]</td>
<td>Store Register Exclusive</td>
<td>-</td>
<td>page 3-36</td>
</tr>
<tr>
<td>STREXB</td>
<td>Rd, Rt, [Rn]</td>
<td>Store Register Exclusive Byte</td>
<td>-</td>
<td>page 3-36</td>
</tr>
<tr>
<td>STREXH</td>
<td>Rd, Rt, [Rn]</td>
<td>Store Register Exclusive Halfword</td>
<td>-</td>
<td>page 3-36</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Operands</td>
<td>Brief description</td>
<td>Flags</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------</td>
<td>---------------------------------------</td>
<td>----------</td>
<td>---------</td>
</tr>
<tr>
<td>STRH, STRHT</td>
<td>Rt, [Rn, #offset]</td>
<td>Store Register Halfword</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>STRT</td>
<td>Rt, [Rn, #offset]</td>
<td>Store Register word</td>
<td>-</td>
<td>page 3-22</td>
</tr>
<tr>
<td>SUB, SUBS</td>
<td>(Rd, ) Rn, Op2</td>
<td>Subtract</td>
<td>N,Z,C,V</td>
<td>page 3-41</td>
</tr>
<tr>
<td>SUB, SUBW</td>
<td>(Rd, ) Rn, #imm12</td>
<td>Subtract</td>
<td>-</td>
<td>page 3-41</td>
</tr>
<tr>
<td>SVC</td>
<td>#imm</td>
<td>Supervisor Call</td>
<td>-</td>
<td>page 3-167</td>
</tr>
<tr>
<td>SXTAB</td>
<td>(Rd, ) Rn, Rm, {, ROR #}</td>
<td>Extend 8 bits to 32 and add</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>SXTAB16</td>
<td>(Rd, ) Rn, Rm, {, ROR #}</td>
<td>Dual extend 8 bits to 16 and add</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>SXTAHI</td>
<td>(Rd, ) Rn, Rm, {, ROR #}</td>
<td>Extend 16 bits to 32 and add</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>SXTB16</td>
<td>(Rd, ) Rm, {, ROR #}</td>
<td>Signed Extend Byte 16</td>
<td>-</td>
<td>page 3-110</td>
</tr>
<tr>
<td>SXTB</td>
<td>(Rd, ) Rm, {, ROR #}</td>
<td>Sign extend a byte</td>
<td>-</td>
<td>page 3-117</td>
</tr>
<tr>
<td>SXTH</td>
<td>(Rd, ) Rm, {, ROR #}</td>
<td>Sign extend a halfword</td>
<td>-</td>
<td>page 3-117</td>
</tr>
<tr>
<td>TBB</td>
<td>[Rn, Rm]</td>
<td>Table Branch Byte</td>
<td>-</td>
<td>page 3-124</td>
</tr>
<tr>
<td>TBH</td>
<td>[Rn, Rm, LSL #1]</td>
<td>Table Branch Halfword</td>
<td>-</td>
<td>page 3-124</td>
</tr>
<tr>
<td>TEQ</td>
<td>Rn, Op2</td>
<td>Test Equivalence</td>
<td>N,Z,C</td>
<td>page 3-62</td>
</tr>
<tr>
<td>TST</td>
<td>Rn, Op2</td>
<td>Test</td>
<td>N,Z,C</td>
<td>page 3-62</td>
</tr>
<tr>
<td>UADD16</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Add 16</td>
<td>GE</td>
<td>page 3-63</td>
</tr>
<tr>
<td>UADD8</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Add 8</td>
<td>GE</td>
<td>page 3-63</td>
</tr>
<tr>
<td>USAX</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Subtract and Add with Exchange</td>
<td>GE</td>
<td>page 3-64</td>
</tr>
<tr>
<td>UHADD16</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Add 16</td>
<td>-</td>
<td>page 3-66</td>
</tr>
<tr>
<td>UHADD8</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Add 8</td>
<td>-</td>
<td>page 3-66</td>
</tr>
<tr>
<td>UHASX</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Add and Subtract with Exchange</td>
<td>-</td>
<td>page 3-67</td>
</tr>
<tr>
<td>UHSAX</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Subtract and Add with Exchange</td>
<td>-</td>
<td>page 3-67</td>
</tr>
<tr>
<td>UHSUB16</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Subtract 16</td>
<td>-</td>
<td>page 3-69</td>
</tr>
<tr>
<td>UHSUBB</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Halving Subtract 8</td>
<td>-</td>
<td>page 3-69</td>
</tr>
<tr>
<td>UBFX</td>
<td>Rd, Rn, #lsb, #width</td>
<td>Unsigned Bit Field Extract</td>
<td>-</td>
<td>page 3-116</td>
</tr>
<tr>
<td>UDIV</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Divide</td>
<td>-</td>
<td>page 3-94</td>
</tr>
<tr>
<td>UMAAL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Unsigned Multiply Accumulate Long (32 x 32 + 32), 64-bit result</td>
<td>-</td>
<td>page 3-77</td>
</tr>
<tr>
<td>UMLAL</td>
<td>RdLo, RdHi, Rn, Rm</td>
<td>Unsigned Multiply with Accumulate (32 x 32 + 64), 64-bit result</td>
<td>-</td>
<td>page 3-93</td>
</tr>
<tr>
<td>UMULL</td>
<td>RdLo, Rdhi, Rn, Rm</td>
<td>Unsigned Multiply (32 x 32), 64-bit result</td>
<td>-</td>
<td>page 3-93</td>
</tr>
<tr>
<td>UQADD16</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Saturating Add 16</td>
<td>-</td>
<td>page 3-105</td>
</tr>
<tr>
<td>UQADD8</td>
<td>(Rd, ) Rn, Rm</td>
<td>Unsigned Saturating Add 8</td>
<td>-</td>
<td>page 3-105</td>
</tr>
</tbody>
</table>
Table 3-1 Cortex-M4 instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>UQASX</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Saturating Add and Subtract with Exchange</td>
<td>-</td>
<td>page 3-103</td>
</tr>
<tr>
<td>UQASX</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Saturating Subtract and Add with Exchange</td>
<td>-</td>
<td>page 3-103</td>
</tr>
<tr>
<td>USUB16</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Saturating Subtract 16</td>
<td>-</td>
<td>page 3-105</td>
</tr>
<tr>
<td>USUB8</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Saturating Subtract 8</td>
<td>-</td>
<td>page 3-105</td>
</tr>
<tr>
<td>USAD8</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Sum of Absolute Differences</td>
<td>-</td>
<td>page 3-71</td>
</tr>
<tr>
<td>USADA8</td>
<td>(Rd,) Rn, Rm, Ra</td>
<td>Unsigned Sum of Absolute Differences and Accumulate</td>
<td>-</td>
<td>page 3-72</td>
</tr>
<tr>
<td>USAT</td>
<td>Rd, #n, Rm {,shift #s}</td>
<td>Unsigned Saturate</td>
<td>Q</td>
<td>page 3-96</td>
</tr>
<tr>
<td>USAT16</td>
<td>Rd, #n, Rm</td>
<td>Unsigned Saturate 16</td>
<td>Q</td>
<td>page 3-97</td>
</tr>
<tr>
<td>UASX</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Add and Subtract with Exchange</td>
<td>GE</td>
<td>page 3-64</td>
</tr>
<tr>
<td>USUB16</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Subtract 16</td>
<td>GE</td>
<td>page 3-73</td>
</tr>
<tr>
<td>USUB8</td>
<td>(Rd,) Rn, Rm</td>
<td>Unsigned Subtract 8</td>
<td>GE</td>
<td>page 3-73</td>
</tr>
<tr>
<td>UXTAB</td>
<td>(Rd,) Rn, Rm{,ROR #}</td>
<td>Rotate, extend 8 bits to 32 and Add</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>UXTAB16</td>
<td>(Rd,) Rn, Rm{,ROR #}</td>
<td>Rotate, dual extend 8 bits to 16 and Add</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>UXTAH</td>
<td>(Rd,) Rn, Rm{,ROR #}</td>
<td>Rotate, unsign extend and Add Halfword</td>
<td>-</td>
<td>page 3-112</td>
</tr>
<tr>
<td>UXTB</td>
<td>(Rd,) Rn{,ROR #}</td>
<td>Zero extend a Byte</td>
<td>-</td>
<td>page 3-117</td>
</tr>
<tr>
<td>UXTB16</td>
<td>(Rd,) Rm{,ROR #}</td>
<td>Unsigned Extend Byte 16</td>
<td>-</td>
<td>page 3-117</td>
</tr>
<tr>
<td>UXTH</td>
<td>(Rd,) Rm{,ROR #}</td>
<td>Zero extend a Halfword</td>
<td>-</td>
<td>page 3-117</td>
</tr>
<tr>
<td>VABS.F32</td>
<td>Sd, Sm</td>
<td>Floating-point Absolute</td>
<td>-</td>
<td>page 3-128</td>
</tr>
<tr>
<td>VADD.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Add</td>
<td>-</td>
<td>page 3-129</td>
</tr>
<tr>
<td>VCMP.F32</td>
<td>Sd, &lt;Sm</td>
<td>#0.0&gt;</td>
<td>Compare two floating-point registers, or one floating-point register and zero</td>
<td>FPSCR</td>
</tr>
<tr>
<td>VCMPE.F32</td>
<td>Sd, &lt;Sm</td>
<td>#0.0&gt;</td>
<td>Compare two floating-point registers, or one floating-point register and zero with invalid operation check</td>
<td>FPSCR</td>
</tr>
<tr>
<td>VCVT.S32.F32</td>
<td>Sd, Sm</td>
<td>Convert between floating-point and integer</td>
<td>-</td>
<td>page 3-131</td>
</tr>
<tr>
<td>VCVT.S16.F32</td>
<td>Sd, Sd, #bits</td>
<td>Convert between floating-point and fixed point</td>
<td>-</td>
<td>page 3-132</td>
</tr>
<tr>
<td>VCVTR.S32.F32</td>
<td>Sd, Sm</td>
<td>Convert between floating-point and integer with rounding</td>
<td>-</td>
<td>page 3-131</td>
</tr>
<tr>
<td>VCVT&lt;8</td>
<td>H&gt;.F32,F16</td>
<td>Sd, Sm</td>
<td>Converts half-precision value to single-precision</td>
<td>-</td>
</tr>
<tr>
<td>VCVTT&lt;8</td>
<td>T&gt;.F32,F16</td>
<td>Sd, Sm</td>
<td>Converts single-precision register to half-precision</td>
<td>-</td>
</tr>
<tr>
<td>VDIV.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Divide</td>
<td>-</td>
<td>page 3-134</td>
</tr>
<tr>
<td>VFMA.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Fused Multiply Accumulate</td>
<td>-</td>
<td>page 3-135</td>
</tr>
<tr>
<td>VFNMA.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Fused Negate Multiply Accumulate</td>
<td>-</td>
<td>page 3-136</td>
</tr>
<tr>
<td>VFMS.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Fused Multiply Subtract</td>
<td>-</td>
<td>page 3-135</td>
</tr>
</tbody>
</table>
### Table 3-1 Cortex-M4 instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operands</th>
<th>Brief description</th>
<th>Flags</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VFNM.S.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Fused Negate Multiply Subtract</td>
<td>-</td>
<td>page 3-136</td>
</tr>
<tr>
<td>VLDM.F&lt;32</td>
<td>64&gt;</td>
<td>Rn[1], list</td>
<td>Load Multiple extension registers</td>
<td>-</td>
</tr>
<tr>
<td>VLDR.F&lt;32</td>
<td>64&gt;</td>
<td>&lt;Dd</td>
<td>Sd&gt;, [Rn]</td>
<td>Load an extension register from memory</td>
</tr>
<tr>
<td>VLMA.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Multiply Accumulate</td>
<td>-</td>
<td>page 3-139</td>
</tr>
<tr>
<td>VLM.S.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Multiply Subtract</td>
<td>-</td>
<td>page 3-139</td>
</tr>
<tr>
<td>VMOV.F32</td>
<td>Sd, #imm</td>
<td>Floating-point Move immediate</td>
<td>-</td>
<td>page 3-140</td>
</tr>
<tr>
<td>VMOV</td>
<td>Sd, Sm</td>
<td>Floating-point Move register</td>
<td>-</td>
<td>page 3-141</td>
</tr>
<tr>
<td>VMOV</td>
<td>Sn, Rt</td>
<td>Copy ARM core register to single precision</td>
<td>-</td>
<td>page 3-143</td>
</tr>
<tr>
<td>VMOV</td>
<td>Sm, Sm1, Rt, Rt2</td>
<td>Copy 2 ARM core registers to 2 single precision</td>
<td>-</td>
<td>page 3-144</td>
</tr>
<tr>
<td>VMOV</td>
<td>Dd[x], Rt</td>
<td>Copy ARM core register to scalar</td>
<td>-</td>
<td>page 3-145</td>
</tr>
<tr>
<td>VMOV</td>
<td>Rt, Dn[x]</td>
<td>Copy scalar to ARM core register</td>
<td>-</td>
<td>page 3-142</td>
</tr>
<tr>
<td>VMSR</td>
<td>Rt, FPSCR</td>
<td>Move FPSCR to ARM core register or APSR</td>
<td>N,Z,C,V</td>
<td>page 3-146</td>
</tr>
<tr>
<td>VMSR</td>
<td>FPSCR, Rt</td>
<td>Move to FPSCR from ARM Core register</td>
<td>FPSCR</td>
<td>page 3-147</td>
</tr>
<tr>
<td>VMUL.F32</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Multiply</td>
<td>-</td>
<td>page 3-148</td>
</tr>
<tr>
<td>VNEG.F32</td>
<td>Sd, Sm</td>
<td>Floating-point Negate</td>
<td>-</td>
<td>page 3-149</td>
</tr>
<tr>
<td>VNM.F32</td>
<td>Sd, Sn, Sm</td>
<td>Floating-point Multiply and Add</td>
<td>-</td>
<td>page 3-150</td>
</tr>
<tr>
<td>VNM.F32</td>
<td>Sd, Sn, Sm</td>
<td>Floating-point Multiply and Subtract</td>
<td>-</td>
<td>page 3-150</td>
</tr>
<tr>
<td>VNMUL</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Multiply</td>
<td>-</td>
<td>page 3-150</td>
</tr>
<tr>
<td>VPOP</td>
<td>list</td>
<td>Pop extension registers</td>
<td>-</td>
<td>page 3-151</td>
</tr>
<tr>
<td>VPUSH</td>
<td>list</td>
<td>Push extension registers</td>
<td>-</td>
<td>page 3-152</td>
</tr>
<tr>
<td>VSQRT.F32</td>
<td>Sd, Sm</td>
<td>Calculates floating-point Square Root</td>
<td>-</td>
<td>page 3-153</td>
</tr>
<tr>
<td>VSTM</td>
<td>Rn[1], list</td>
<td>Floating-point register Store Multiple</td>
<td>-</td>
<td>page 3-154</td>
</tr>
<tr>
<td>VSTR.F&lt;32</td>
<td>64&gt;</td>
<td>Sd, [Rn]</td>
<td>Stores an extension register to memory</td>
<td>-</td>
</tr>
<tr>
<td>VSUB.F&lt;32</td>
<td>64&gt;</td>
<td>(Sd,) Sn, Sm</td>
<td>Floating-point Subtract</td>
<td>-</td>
</tr>
<tr>
<td>WFE</td>
<td>-</td>
<td>Wait For Event</td>
<td>-</td>
<td>page 3-168</td>
</tr>
<tr>
<td>WFI</td>
<td>-</td>
<td>Wait For Interrupt</td>
<td>-</td>
<td>page 3-169</td>
</tr>
</tbody>
</table>
3.2 CMSIS functions

ISO/IEC C code cannot directly access some Cortex-M4 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>
<thead>
<tr>
<th>Instruction</th>
<th>CMSIS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPSIE I</td>
<td>void __enable_irq(void)</td>
</tr>
<tr>
<td>CPSID I</td>
<td>void __disable_irq(void)</td>
</tr>
<tr>
<td>CPSIE F</td>
<td>void __enable_fault_irq(void)</td>
</tr>
<tr>
<td>CPSID F</td>
<td>void __disable_fault_irq(void)</td>
</tr>
<tr>
<td>ISB</td>
<td>void __ISB(void)</td>
</tr>
<tr>
<td>DSB</td>
<td>void __DSB(void)</td>
</tr>
<tr>
<td>DMB</td>
<td>void __DMB(void)</td>
</tr>
<tr>
<td>REV</td>
<td>uint32_t __REV(uint32_t int value)</td>
</tr>
<tr>
<td>REV16</td>
<td>uint32_t __REV16(uint32_t int value)</td>
</tr>
<tr>
<td>REVSH</td>
<td>uint32_t __REVSH(uint32_t int value)</td>
</tr>
<tr>
<td>RBIT</td>
<td>uint32_t __RBIT(uint32_t int value)</td>
</tr>
<tr>
<td>SEV</td>
<td>void __SEV(void)</td>
</tr>
<tr>
<td>WFE</td>
<td>void __WFE(void)</td>
</tr>
<tr>
<td>WFI</td>
<td>void __WFI(void)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Special register</th>
<th>Access</th>
<th>CMSIS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMASK</td>
<td>Read</td>
<td>uint32_t __get_PRIMASK (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_PRIMASK (uint32_t value)</td>
</tr>
<tr>
<td>FAULTMASK</td>
<td>Read</td>
<td>uint32_t __get_FAULTMASK (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_FAULTMASK (uint32_t value)</td>
</tr>
<tr>
<td>BASEPRI</td>
<td>Read</td>
<td>uint32_t __get_BASEPRI (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_BASEPRI (uint32_t value)</td>
</tr>
<tr>
<td>CONTROL</td>
<td>Read</td>
<td>uint32_t __get_CONTROL (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_CONTROL (uint32_t value)</td>
</tr>
</tbody>
</table>
## Table 3-3 CMSIS functions to access the special registers (continued)

<table>
<thead>
<tr>
<th>Special register</th>
<th>Access</th>
<th>CMSIS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP</td>
<td>Read</td>
<td>uint32_t __get_MSP (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_MSP (uint32_t TopOfMainStack)</td>
</tr>
<tr>
<td>PSP</td>
<td>Read</td>
<td>uint32_t __get_PSP (void)</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>void __set_PSP (uint32_t TopOfProcStack)</td>
</tr>
</tbody>
</table>
3.3 About the instruction descriptions

The following sections give more information about using the instructions:

• *Operands* on page 3-12
• *Restrictions when using PC or SP* on page 3-12
• *Flexible second operand* on page 3-12
• *Shift Operations* on page 3-13
• *Address alignment* on page 3-17
• *PC-relative expressions* on page 3-17
• *Conditional execution* on page 3-18
• *Instruction width selection* on page 3-21.
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-M4 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 3-13

**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 the constants shown above, X and Y are hexadecimal digits.

——— Note ————

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 MOV, 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:

\( Rm \) The register holding the data for the second operand.

\( shift \) An optional shift to be applied to \( Rm \). It can be one of:

- \( \text{ASR} \ #n \) Arithmetic shift right \( n \) bits, \( 1 \leq n \leq 32 \).
- \( \text{LSL} \ #n \) Logical shift left \( n \) bits, \( 1 \leq n \leq 31 \).
- \( \text{LSR} \ #n \) Logical shift right \( n \) bits, \( 1 \leq n \leq 32 \).
- \( \text{ROR} \ #n \) Rotate right \( n \) bits, \( 1 \leq n \leq 31 \).
- \( \text{RRX} \) Rotate right one bit, with extend.

If you omit the shift, or specify \( \text{LSL} \ #0 \), 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 \( Rm \) remains 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 3-12. 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 3-12. 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 3-14.
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 \( Rm \).

**Fig. 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.

**Fig. 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 3-15.
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 the 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\([0]\) of the result. See Figure 3-5 on page 3-16.

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-M4 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 2-29.

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 4-19.

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 3-19 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 3-122 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 3-19
• Condition code suffixes on page 3-19.
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 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*.

A carry occurs:
- if the result of an addition is greater than or equal to $2^{32}$
- if the result of a subtraction is positive or zero
- as the result of an inline barrel shifter operation in a move or logical instruction.

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>
<thead>
<tr>
<th>Suffix</th>
<th>Flags</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>Z = 1</td>
<td>Equal</td>
</tr>
<tr>
<td>NE</td>
<td>Z = 0</td>
<td>Not equal</td>
</tr>
<tr>
<td>CS or HS</td>
<td>C = 1</td>
<td>Higher or same, unsigned</td>
</tr>
<tr>
<td>CC or LO</td>
<td>C = 0</td>
<td>Lower, unsigned</td>
</tr>
<tr>
<td>MI</td>
<td>N = 1</td>
<td>Negative</td>
</tr>
<tr>
<td>PL</td>
<td>N = 0</td>
<td>Positive or zero</td>
</tr>
</tbody>
</table>
Example 3-1 shows the use of a conditional instruction to find the absolute value of a number. $R0 = \text{abs}(R1)$.

**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>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>Generate PC-relative address</td>
<td>ADR on page 3-23</td>
</tr>
<tr>
<td>CLREX</td>
<td>Clear Exclusive</td>
<td>CLREX on page 3-38</td>
</tr>
<tr>
<td>LDM{mode}</td>
<td>Load Multiple registers</td>
<td>LDM and STM on page 3-32</td>
</tr>
<tr>
<td>LDR{type}</td>
<td>Load Register using immediate offset</td>
<td>LDR and STR, immediate offset on page 3-24</td>
</tr>
<tr>
<td>LDR{type}</td>
<td>Load Register using register offset</td>
<td>LDR and STR, register offset on page 3-27</td>
</tr>
<tr>
<td>LDR{type}T</td>
<td>Load Register with unprivileged access</td>
<td>LDR and STR, unprivileged on page 3-29</td>
</tr>
<tr>
<td>LDR</td>
<td>Load Register using PC-relative address</td>
<td>LDR, PC-relative on page 3-30</td>
</tr>
<tr>
<td>LDREX{type}</td>
<td>Load Register Exclusive</td>
<td>LDREX and STREX on page 3-36</td>
</tr>
<tr>
<td>POP</td>
<td>Pop registers from stack</td>
<td>PUSH and POP on page 3-34</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push registers onto stack</td>
<td>PUSH and POP on page 3-34</td>
</tr>
<tr>
<td>STM{mode}</td>
<td>Store Multiple registers</td>
<td>LDM and STM on page 3-32</td>
</tr>
<tr>
<td>STR{type}</td>
<td>Store Register using immediate offset</td>
<td>LDR and STR, immediate offset on page 3-24</td>
</tr>
<tr>
<td>STR{type}</td>
<td>Store Register using register offset</td>
<td>LDR and STR, register offset on page 3-27</td>
</tr>
<tr>
<td>STR{type}T</td>
<td>Store Register with unprivileged access</td>
<td>LDR and STR, unprivileged on page 3-29</td>
</tr>
<tr>
<td>STREX{type}</td>
<td>Store Register Exclusive</td>
<td>LDREX and STREX on page 3-36</td>
</tr>
</tbody>
</table>
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 3-18*.
- **Rd**  Specifies the destination register.
- **label**  Is a PC-relative expression. See *PC-relative expressions on page 3-17*.

**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 to 1 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 3-21*.

---

**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**

\[
\begin{align*}
\text{Op}\{\text{type}\}\{\text{cond}\} & \ R_t, [R_n \{, \#\text{offset}] \quad ; \text{immediate offset} \\
\text{Op}\{\text{type}\}\{\text{cond}\} & \ R_t, [R_n, \#\text{offset}]! \quad ; \text{pre-indexed} \\
\text{Op}\{\text{type}\}\{\text{cond}\} & \ R_t, [R_n, \#\text{offset}] \quad ; \text{post-indexed} \\
\text{OpD}\{\text{cond}\} & \ R_t, R_{t2}, [R_n \{, \#\text{offset}] \quad ; \text{immediate offset, two words} \\
\text{OpD}\{\text{cond}\} & \ R_t, R_{t2}, [R_n, \#\text{offset}]! \quad ; \text{pre-indexed, two words} \\
\text{OpD}\{\text{cond}\} & \ R_t, R_{t2}, [R_n, \#\text{offset}] \quad ; \text{post-indexed, two words}
\end{align*}
\]

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 3-18*.
- **Rt** Specifies the register to load or store.
- **Rn** Specifies the register on which the memory address is based.
- **offset** Specifies an offset from Rn. If offset is omitted, the address is the contents of Rn.
- **Rt2** Specifies 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:

\[
[Rn, \#\text{offset}]
\]

**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:

\[
[Rn, \#\text{offset}]
\]

**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:

\[
[Rn], \#\text{offset}
\]

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 3-17*.

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

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Immediate offset</th>
<th>Pre-indexed</th>
<th>Post-indexed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word, halfword, signed halfword, byte, or signed byte</td>
<td>−255 to 4095</td>
<td>−255 to 255</td>
<td>−255 to 255</td>
</tr>
<tr>
<td>Two words</td>
<td>multiple of 4 in the range −1020 to 1020</td>
<td>multiple of 4 in the range −1020 to 1020</td>
<td>multiple of 4 in the range −1020 to 1020</td>
</tr>
</tbody>
</table>
Restrictions

For load instructions:
• \( R_t \) can be SP or PC for word loads only
• \( R_t \) must be different from \( R_t2 \) for two-word loads
• \( R_n \) must be different from \( R_t \) and \( R_t2 \) in the pre-indexed or post-indexed forms.

When \( R_t \) 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:
• \( R_t \) can be SP for word stores only
• \( R_t \) must not be PC
• \( R_n \) must not be PC
• \( R_n \) must be different from \( R_t \) and \( R_t2 \) in the pre-indexed or post-indexed forms.

Condition flags

These instructions do not change the flags.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR R8, [R10]</td>
<td>Loads R8 from the address in R10.</td>
</tr>
<tr>
<td>LDRNE R2, [R5, #960]!</td>
<td>Loads (conditionally) R2 from a word 960 bytes above the address in R5, and increments R5 by 960.</td>
</tr>
<tr>
<td>STR R2, [R9,#const-struc]</td>
<td>Stores R2 as halfword data into address in R9, and increments R9 by the value of const-struc.</td>
</tr>
<tr>
<td>STRH R3, [R4], #4</td>
<td>Stores R3 as halfword data into address in R4, and increments R4 by 4.</td>
</tr>
<tr>
<td>LDRD R8, R9, [R3, #0x20]</td>
<td>Loads R8 from a word 32 bytes above the address in R3, and loads R9 from a word 36 bytes above the address in R3.</td>
</tr>
<tr>
<td>STRD R0, R1, [R8], #-16</td>
<td>Stores R0 to address in R8, and store R1 to a word 4 bytes above the address in R8, and then decrement R8 by 16.</td>
</tr>
</tbody>
</table>
3.4.3 LDR and STR, register offset

Load and Store with register offset.

Syntax

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

where:

\text{op} \quad \text{Is one of:}
\begin{align*}
\text{LDR} & \quad \text{Load Register.} \\
\text{STR} & \quad \text{Store Register.}
\end{align*}

\text{type} \quad \text{Is one of:}
\begin{align*}
\text{B} & \quad \text{unsigned byte, zero extend to 32 bits on loads.} \\
\text{SB} & \quad \text{signed byte, sign extend to 32 bits (LDR only).} \\
\text{H} & \quad \text{unsigned halfword, zero extend to 32 bits on loads.} \\
\text{SH} & \quad \text{signed halfword, sign extend to 32 bits (LDR only).} \\
\text{-} & \quad \text{omit, for word.}
\end{align*}

\text{cond} \quad \text{Is an optional condition code, see } \text{Conditional execution on page 3-18.}

\text{Rt} \quad \text{Specifies the register to load or store.}

\text{Rn} \quad \text{Specifies the register on which the memory address is based.}

\text{Rm} \quad \text{Specifies the register containing a value to be used as the offset.}

\text{LSL \ #n} \quad \text{Is an optional shift, with } n \text{ in the range 0 to 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 \( Rn \) 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 \text{Address alignment on page 3-17.}

Restrictions

In these instructions:

\begin{itemize}
\item \( Rn \) must not be PC
\item \( Rm \) must not be SP and must not be PC
\item \( Rt \) can be SP only for word loads and word stores
\item \( Rt \) can be PC only for word loads.
\end{itemize}

When \( Rt \) is PC in a word load instruction:

\begin{itemize}
\item bit[0] of the loaded value must be 1 for correct execution, and a branch occurs to this

halfword-aligned address
\item if the instruction is conditional, it must be the last instruction in the IT block.
\end{itemize}
Condition flags

These instructions do not change the flags.

Examples

```assembly
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

\[ \text{op}\{\text{type}\}T{\text{cond}} \text{ Rt, [Rn \{, \#offset\}] ; immediate offset} \]

where:

\( \text{op} \)

Is one of:

- \text{LDR} Load Register.
- \text{STR} Store Register.

\( \text{type} \)

Is one of:

- \text{B} unsigned byte, zero extend to 32 bits on loads.
- \text{SB} signed byte, sign extend to 32 bits (LDR only).
- \text{H} unsigned halfword, zero extend to 32 bits on loads.
- \text{SH} signed halfword, sign extend to 32 bits (LDR only).
- - omit, for word.

\( \text{cond} \)

Is an optional condition code, see Conditional execution on page 3-18.

\( \text{Rt} \)

Specifies the register to load or store.

\( \text{Rn} \)

Specifies the register on which the memory address is based.

\( \text{offset} \)

Specifies an offset from \( \text{Rn} \) and can be 0 to 255. If \( \text{offset} \) is omitted, the address is the value in \( \text{Rn} \).

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 3-24. 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:

- \( \text{Rn} \) must not be PC
- \( \text{Rt} \) must not be SP and must not be PC.

Condition flags

These instructions do not change the flags.

Examples

\[ \text{STRBTEQ R4, [R7]} \quad ; \text{Conditionally store least significant byte in} \]
\[ \quad \quad \text{R4 to an address in R7, with unprivileged access} \]
\[ \text{LDRHT R2, [R2, \#8]} \quad ; \text{Load halfword value from an address equal to} \]
\[ \quad \quad \text{sum of R2 and 8 into R2, with unprivileged access}. \]
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 3-18.

- **Rt** Specifies the register to load or store.

- **Rt2** Specifies the second register to load or store.

- **label** Is a PC-relative expression. See PC-relative expressions on page 3-17.

**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 3-17.

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

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Offset range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word, halfword, signed halfword, byte, signed byte</td>
<td>−4095 to 4095</td>
</tr>
<tr>
<td>Two words</td>
<td>−1020 to 1020</td>
</tr>
</tbody>
</table>

**Note**

You might have to use the .W suffix to get the maximum offset range. See Instruction width selection on page 3-21.

**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

\[ \text{op\{addr\_mode\}}\{\text{cond}\} \text{ Rn[!], reglist} \]

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 3-18.

- **Rn** specifies 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 3-33.

LDM and LDMFD are synonyms for LDMIA. 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.

STM and STMEA are synonyms for STMIA. 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 happen 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 3-34 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 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 3-18.
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 3-32 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.
3.4.8 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 3-18.
Rd Specifies the destination register for the returned status.
Rt Specifies the register to load or store.
Rn Specifies 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 2-18.

If an 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?
   IIT  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
.... try                         ; Yes - we have the lock.
```
3.4.9 CLREX

Clear Exclusive.

**Syntax**

CLREX\{cond\}

where:

\textit{cond} Is an optional condition code, see \textit{Conditional execution on page 3-18}.

**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 \textit{Synchronization primitives on page 2-18} 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>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td>$ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>$ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>ADDW</td>
<td>Add</td>
<td>$ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
<td>$AND, ORR, EOR, BIC, and ORN on page 3-44</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
<td>$ASR, LSL, LSR, ROR, and RRX on page 3-46</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit Clear</td>
<td>$AND, ORR, EOR, BIC, and ORN on page 3-44</td>
</tr>
<tr>
<td>CLZ</td>
<td>Count leading zeros</td>
<td>CLZ on page 3-48</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>CMP and CMN on page 3-49</td>
</tr>
<tr>
<td>EOR</td>
<td>Exclusive OR</td>
<td>$AND, ORR, EOR, BIC, and ORN on page 3-44</td>
</tr>
<tr>
<td>LSL</td>
<td>Logical Shift Left</td>
<td>$ASR, LSL, LSR, ROR, and RRX on page 3-46</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical Shift Right</td>
<td>$ASR, LSL, LSR, ROR, and RRX on page 3-46</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>MOV and MVN on page 3-50</td>
</tr>
<tr>
<td>MOVT</td>
<td>Move Top</td>
<td>MOVT on page 3-52</td>
</tr>
<tr>
<td>MOVW</td>
<td>Move 16-bit constant</td>
<td>MOV and MVN on page 3-50</td>
</tr>
<tr>
<td>MVN</td>
<td>Move NOT</td>
<td>MOV and MVN on page 3-50</td>
</tr>
<tr>
<td>ORN</td>
<td>Logical OR NOT</td>
<td>$AND, ORR, EOR, BIC, and ORN on page 3-44</td>
</tr>
<tr>
<td>ORR</td>
<td>Logical OR</td>
<td>$AND, ORR, EOR, BIC, and ORN on page 3-44</td>
</tr>
<tr>
<td>RBIT</td>
<td>Reverse Bits</td>
<td>REV, REV16, REVSX, and RBIT on page 3-53</td>
</tr>
<tr>
<td>REV</td>
<td>Reverse byte order in a word</td>
<td>REV, REV16, REVSX, and RBIT on page 3-53</td>
</tr>
<tr>
<td>REV16</td>
<td>Reverse byte order in each halfword</td>
<td>REV, REV16, REVSX, and RBIT on page 3-53</td>
</tr>
<tr>
<td>REVSX</td>
<td>Reverse byte order in bottom halfword and sign extend</td>
<td>REV, REV16, REVSX, and RBIT on page 3-53</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate Right</td>
<td>$ASR, LSL, LSR, ROR, and RRX on page 3-46</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate Right with Extend</td>
<td>$ASR, LSL, LSR, ROR, and RRX on page 3-46</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse Subtract</td>
<td>$ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>SADD16</td>
<td>Signed Add 16</td>
<td>SADD16 and SADD8 on page 3-54</td>
</tr>
<tr>
<td>SADD8</td>
<td>Signed Add 8</td>
<td>SADD16 and SADD8 on page 3-54</td>
</tr>
<tr>
<td>SASX</td>
<td>Signed Add and Subtract with Exchange</td>
<td>SASX and SSAX on page 3-60</td>
</tr>
<tr>
<td>SSAX</td>
<td>Signed Subtract and Add with Exchange</td>
<td>SASX and SSAX on page 3-60</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with Carry</td>
<td>$ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>Mnemonic</td>
<td>Brief description</td>
<td>See</td>
</tr>
<tr>
<td>----------</td>
<td>------------------</td>
<td>-----</td>
</tr>
<tr>
<td>SHADD16</td>
<td>Signed Halving Add 16</td>
<td>SHADD16 and SHADD8 on page 3-55</td>
</tr>
<tr>
<td>SHADD8</td>
<td>Signed Halving Add 8</td>
<td>SHADD16 and SHADD8 on page 3-55</td>
</tr>
<tr>
<td>SHASX</td>
<td>Signed Halving Add and Subtract with Exchange</td>
<td>SHASX and SHSAX on page 3-56</td>
</tr>
<tr>
<td>SHSAX</td>
<td>Signed Halving Subtract and Add with Exchange</td>
<td>SHASX and SHSAX on page 3-56</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>SUBW</td>
<td>Subtract</td>
<td>ADD, ADC, SUB, SBC, and RSB on page 3-41</td>
</tr>
<tr>
<td>TEQ</td>
<td>Test Equivalence</td>
<td>TST and TEQ on page 3-62</td>
</tr>
<tr>
<td>TST</td>
<td>Test</td>
<td>TST and TEQ on page 3-62</td>
</tr>
<tr>
<td>UADD16</td>
<td>Unsigned Add 16</td>
<td>UADD16 and UADD8 on page 3-63</td>
</tr>
<tr>
<td>UADD8</td>
<td>Unsigned Add 8</td>
<td>UADD16 and UADD8 on page 3-63</td>
</tr>
<tr>
<td>UASX</td>
<td>Unsigned Add and Subtract with Exchange</td>
<td>UASX and USAX on page 3-64</td>
</tr>
<tr>
<td>USAX</td>
<td>Unsigned Subtract and Add with Exchange</td>
<td>UASX and USAX on page 3-64</td>
</tr>
<tr>
<td>UHADD16</td>
<td>Unsigned Halving Add 16</td>
<td>UHADD16 and UHADD8 on page 3-66</td>
</tr>
<tr>
<td>UHADD8</td>
<td>Unsigned Halving Add 8</td>
<td>UHADD16 and UHADD8 on page 3-66</td>
</tr>
<tr>
<td>UHASX</td>
<td>Unsigned Halving Add and Subtract with Exchange</td>
<td>UHASX and UHSAX on page 3-67</td>
</tr>
<tr>
<td>UHASX</td>
<td>Unsigned Halving Subtract and Add with Exchange</td>
<td>UHASX and UHSAX on page 3-67</td>
</tr>
<tr>
<td>UHSUB16</td>
<td>Unsigned Halving Subtract 16</td>
<td>UHSUB16 and UHSUB8 on page 3-69</td>
</tr>
<tr>
<td>UHSUB8</td>
<td>Unsigned Halving Subtract 8</td>
<td>UHSUB16 and UHSUB8 on page 3-69</td>
</tr>
<tr>
<td>USAD8</td>
<td>Unsigned Sum of Absolute Differences</td>
<td>USAD8 on page 3-71</td>
</tr>
<tr>
<td>USAD8</td>
<td>Unsigned Sum of Absolute Differences and Accumulate</td>
<td>USAD8 on page 3-72</td>
</tr>
<tr>
<td>USUB16</td>
<td>Unsigned Subtract 16</td>
<td>USUB16 and USUB8 on page 3-73</td>
</tr>
<tr>
<td>USUB8</td>
<td>Unsigned Subtract 8</td>
<td>USUB16 and USUB8 on page 3-73</td>
</tr>
</tbody>
</table>
3.5.1 ADD, ADC, SUB, SBC, and RSB

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

Syntax

\[
\text{op}(\{S\})\{\text{cond}\} \{\text{Rd},\} \text{Rn}, \text{Operand2}
\]

\[
\text{op}\{\text{cond}\} \{\text{Rd},\} \text{Rn}, \#\text{imm12} \quad ; \text{ADD and SUB only}
\]

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 3-18.

- **cond** Is an optional condition code, see *Conditional execution* on page 3-18.

- **Rd** Specifies the destination register. If Rd is omitted, the destination register is Rn.

- **Rn** Specifies the register holding the first operand.

- **Operand2** Is a flexible second operand. See *Flexible second operand* on page 3-12 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 3-42.

See also *ADR* on page 3-23.

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 to 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

Example 3-4 on page 3-43 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.
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBS</td>
<td>R6, R9</td>
<td>subtract the least significant words</td>
</tr>
<tr>
<td>SBCS</td>
<td>R9, R1</td>
<td>subtract the middle words with carry</td>
</tr>
<tr>
<td>SBC</td>
<td>R2, R11</td>
<td>subtract the most significant words with carry</td>
</tr>
</tbody>
</table>
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 3-18.

\( cond \) Is an optional condition code, see *Conditional execution* on page 3-18.

\( Rd \) Specifies the destination register.

\( Rn \) Specifies the register holding the first operand.

\( Operand2 \) Is a flexible second operand. See *Flexible second operand* on page 3-12 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 3-12
- 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, #0x18
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

\[ \text{op}\{\text{S}\}\{\text{cond}\} \quad R_d, \ R_m, \ R_s \]

\[ \text{op}\{\text{S}\}\{\text{cond}\} \quad R_d, \ R_m, \ #n \]

\[ \text{RRX}\{\text{S}\}\{\text{cond}\} \quad R_d, \ R_m \]

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 3-18.

- **Rd** specifies the destination register.

- **Rm** specifies the register holding the value to be shifted.

- **Rs** specifies the register holding the shift length to apply to the value in \( R_m \). Only the least significant byte is used and can be in the range 0 to 255.

- **n** specifies 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 ---

MOV S Rd, Rm is the preferred syntax for LSL S 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 3-13.

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
- the C flag is updated to the last bit shifted out, except when the shift length is 0, see Shift Operations on page 3-13.

Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASR</td>
<td>R7, R8, #9</td>
<td>Arithmetic shift right by 9 bits</td>
</tr>
<tr>
<td>LSLS</td>
<td>R1, R2, #3</td>
<td>Logical shift left by 3 bits with flag update</td>
</tr>
<tr>
<td>LSR</td>
<td>R4, R5, #6</td>
<td>Logical shift right by 6 bits</td>
</tr>
<tr>
<td>ROR</td>
<td>R4, R5, R6</td>
<td>Rotate right by the value in the bottom byte of R6</td>
</tr>
<tr>
<td>RRX</td>
<td>R4, R5</td>
<td>Rotate right with extend.</td>
</tr>
</tbody>
</table>
3.5.4 CLZ

Count Leading Zeros.

Syntax

\texttt{CLZ\{cond\} Rd, Rm}

where:

\texttt{cond} \quad \text{Is an optional condition code, see \textit{Conditional execution} on page 3-18.}
\texttt{Rd} \quad \text{Specifies the destination register.}
\texttt{Rm} \quad \text{Specifies the operand register.}

Operation

The CLZ instruction counts the number of leading zeros in the value in \texttt{Rm} and returns the result in \texttt{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

\begin{verbatim}
    CLZ    R4, R9
    CLZNE R2, R3
\end{verbatim}
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 3-18.
Rn Specifies the register holding the first operand.
Operand2 Is a flexible second operand. See Flexible second operand on page 3-12 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

\[
\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ \text{Operand2} \\
\text{MOV}\{\text{cond}\} \ R_d, \ #\text{imm16} \\
\text{MVN}\{S\}\{\text{cond}\} \ R_d, \ \text{Operand2}
\]

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 3-18.
- \(\text{cond}\) is an optional condition code. See Conditional execution on page 3-18.
- \(R_d\) specifies the destination register.
- \(\text{Operand2}\) is a flexible second operand, see Flexible second operand on page 3-12 for details of the options.
- \(\text{imm16}\) is any value in the range 0-65535.

Operation

The MOV instruction copies the value of \(\text{Operand2}\) into \(R_d\).

When \(\text{Operand2}\) in a MOV instruction is a register with a shift other than LSL #0, the preferred syntax is the corresponding shift instruction:

- \(\text{ASR}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ #n\) is the preferred syntax for MOV\{S\}\{cond\} \ R_d, \ R_m, \ ASR #n
- \(\text{LSL}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ #n\) is the preferred syntax for MOV\{S\}\{cond\} \ R_d, \ R_m, \ LSL #n\ if n != 0
- \(\text{LSR}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ #n\) is the preferred syntax for MOV\{S\}\{cond\} \ R_d, \ R_m, \ LSR #n
- \(\text{ROR}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ #n\) is the preferred syntax for MOV\{S\}\{cond\} \ R_d, \ R_m, \ ROR #n
- \(\text{RRX}\{S\}\{\text{cond}\} \ R_d, \ R_m\) is the preferred syntax for MOV\{S\}\{cond\} \ R_d, \ R_m, \ RRX.

Also, the MOV instruction permits additional forms of \(\text{Operand2}\) as synonyms for shift instructions:

- \(\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ \text{ASR} \ R_s\) is a synonym for ASR\{S\}\{cond\} \ R_d, \ R_m, \ \text{ASR} \ R_s
- \(\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ \text{LSL} \ R_s\) is a synonym for LSL\{S\}\{cond\} \ R_d, \ R_m, \ \text{LSL} \ R_s
- \(\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ \text{LSR} \ R_s\) is a synonym for LSR\{S\}\{cond\} \ R_d, \ R_m, \ \text{LSR} \ R_s
- \(\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ \text{ROR} \ R_s\) is a synonym for ROR\{S\}\{cond\} \ R_d, \ R_m, \ \text{ROR} \ R_s
- \(\text{MOV}\{S\}\{\text{cond}\} \ R_d, \ R_m, \ \text{RRX} \ R_s\) is a synonym for RRX\{S\}\{cond\} \ R_d, \ R_m, \ \text{RRX}.

See ASR, LSL, LSR, ROR, and RRX on page 3-46.

The MVN instruction takes the value of \(\text{Operand2}\), performs a bitwise logical NOT operation on the value, and places the result into \(R_d\).

---

Note

The MOV\{s\} instruction provides the same function as MOV, but is restricted to using the \textit{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 3-12
• 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

\[
\text{MOVT}\{\text{cond}\}; \text{Rd}, \#\text{imm16}
\]

where:

- \text{cond} is an optional condition code, see \textit{Conditional execution on page 3-18}.
- \text{Rd} specifies the destination register.
- \text{imm16} is a 16-bit immediate constant.

Operation

MOVT writes a 16-bit immediate value, \text{imm16}, to the top halfword, \text{Rd}[31:16], of its destination register. The write does not affect \text{Rd}[15:0].

The MOV, MOVT instruction pair enables you to generate any 32-bit constant.

Restrictions

\text{Rd} must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

\[
\text{MOVT R3, } \#0xF123 ; \text{Write } 0xF123 \text{ to upper halfword of R3, lower halfword } \\
\text{; and APSR are unchanged.}
\]
3.5.8  REV, REV16, REVSH, and RBIT

Reverse bytes and Reverse bits.

Syntax

\[ \text{op}\{\text{cond}\} \ Rd, \ Rn \]

where:

- \text{op}  Is any 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.

- \text{cond}  Is an optional condition code, see \textit{Conditional execution} on page 3-18.

- \text{Rd}  Specifies the destination register.

- \text{Rn}  Specifies 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**

```
{cond}\{Rd,\}  \text{op}  \text{Rn, Rm}
```

where:

- **op** is any 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 3-18](#).

- **Rd** specifies the destination register.

- **Rn** specifies the first register holding the operand.

- **Rm** specifies the second register holding the operand.

**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 do not change the flags.

**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 any of:
  - SHADD16 Signed Halving Add 16
  - SHADD8 Signed Halving Add 8

- \( cond \) Is an optional condition code, see Conditional execution on page 3-18.

- \( Rd \): Specifies the destination register.

- \( Rn \): Specifies the first operand register.

- \( Rm \): Specifies the second operand register.

Operation

Use these instructions to add 16-bit and 8-bit data and then 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

\[ \text{op}\{\text{cond}\} \{Rd\}, \ Rn, \ Rm \]

where:

- \( \text{op} \) Is any of:
  - SHASX Add and Subtract with Exchange and Halving.
  - SHSAX Subtract and Add with Exchange and Halving.

- \( \text{cond} \) Is an optional condition code, see Conditional execution on page 3-18.

- \( Rd \) Specifies the destination register.

- \( Rn, \ Rm \) Are registers holding the first and second operands.

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

\[
op\{\text{cond}\}\{Rd,\} Rn, Rm
\]

where:

- \(op\) is any of:
  - \text{SHSUB16}  Signed Halving Subtract 16
  - \text{SHSUB8}  Signed Halving Subtract 8

- \text{cond} is an optional condition code, see *Conditional execution on page 3-18*.

- \(Rd\) specifies the destination register.

- \(Rn\) specifies the first operand register.

- \(Rm\) specifies 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 SHSUB8 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

- \text{SHSUB16} R1, R0  ; Subtracts halfwords in R0 from corresponding halfword of R1 and writes to corresponding halfword of R1
- \text{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 any 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 3-18*.

- **Rd** Specifies the destination register.

- **Rn** Specifies the first operand register.

- **Rm** Specifies 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 do not change the flags.

**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

\[ \text{op}\{\text{cond}\} \{Rd\}, \ Rm, \ Rn \]

where:

- **op** Is any 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 3-18](#).

- **Rd** Specifies the destination register.

- **Rn**, **Rm** Are registers holding the first and second operands.

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 highword 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 highword 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 do not affect the condition code flags.
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:

Rn Specifies the register holding the first operand.
Operand2 Is a flexible second operand. See Flexible second operand on page 3-12 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 3-12
• 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

\[ \text{op}\{\text{cond}\}\{Rd,\} \ Rn, \ Rm \]

where:

- \( \text{op} \) is any of:
  - UADD16: Performs two 16-bit unsigned integer additions.
  - UADD8: Performs four 8-bit unsigned integer additions.

- \( \text{cond} \) is an optional condition code, see Conditional execution on page 3-18.

- \( Rd \) specifies the destination register.

- \( Rn \) specifies the first register holding the operand.

- \( Rm \) specifies the second register holding the operand.

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 UADD16 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 do not change the flags.

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

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

Syntax

\[ \text{op}\{\text{cond}\} \{Rd\}, Rn, Rm \]

where:

- \text{op} Is one of:
  - \text{UASX} Add and Subtract with Exchange.
  - \text{USAX} Subtract and Add with Exchange.

- \text{cond} Is an optional condition code, see Conditional execution on page 3-18.

- \text{Rd} Specifies the destination register.
- \text{Rn, Rm} Are registers holding the first and second operands.

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 do not affect the condition code flags.
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**

\[ \text{op}\{\text{cond}\}\{\text{Rd},\} \ Rn, \ Rm \]

where:

- **op** Is any of:
  - **UHADD16**  Unsigned Halving Add 16.
  - **UHADD8**  Unsigned Halving Add 8.

- **cond** Is an optional condition code, see *Conditional execution on page 3-18.*

- **Rd** Specifies the destination register.
- **Rn** Specifies the register holding the first operand.
- **Rm** Specifies 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

\[ \text{op}\{\text{cond}\}\{Rd\}, Rn, Rm \]

where:

- \text{op} is one of:
  - \text{UHASX}: Add and Subtract with Exchange and Halving.
  - \text{UHSAX}: Subtract and Add with Exchange and Halving.
- \text{cond} is an optional condition code, see \textit{Conditional execution} on page 3-18.
- \text{Rd} specifies the destination register.
- \text{Rn}, \text{Rm} are registers holding the first and second operands.

Operation

The \text{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 highword 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 division in the bottom halfword of the destination register.

The \text{UHSAX} instruction:
1. Subtracts the bottom halfword of the second operand from the top highword 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

\[
\text{op}(\text{cond}); \{Rd,\} \ Rn, \ Rm
\]

where:

- \text{op} is any of:
  - \text{UHSUB16} performs two unsigned 16-bit integer additions, halves the results, and writes the results to the destination register.
  - \text{UHSUB8} performs four unsigned 8-bit integer additions, halves the results, and writes the results to the destination register.

- \text{cond} is an optional condition code, see Conditional execution on page 3-18.

- \text{Rd} specifies the destination register.

- \text{Rn} specifies the first register holding the operand.

- \text{Rm} specifies the second register holding the operand.

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

\[
\begin{align*}
\text{UHSUB16} & \quad \text{R1, R0} \quad ; \text{Subtracts halfwords in R0 from corresponding halfword of} \\
& \quad \text{R1 and writes halved result to corresponding halfword in} \\
& \quad \text{R1} \\
\text{UHSUB8} & \quad \text{R4, R0, R5} \quad ; \text{Subtracts bytes of R5 from corresponding byte in R0 and} \\
& \quad \text{writes halved result to corresponding byte in R4.}
\end{align*}
\]
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{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

where:
- `<c>`, `<q>` is a standard assembler syntax fields.
- `<Rd>` specifies the destination register.
- `<Rn>` specifies the first operand register.
- `<Rm>` specifies 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.

**Restrictions**

None.

**Condition flags**

These instructions do not change the flags.

**Examples**

SADD16 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 3-18.
Rd Specifies the destination register.
Rn Specifies the first operand register.
Rm Specifies 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:

- \textit{cond} Is an optional condition code, see Conditional execution on page 3-18.
- \textit{Rd} Specifies the destination register.
- \textit{Rn} Specifies the first operand register.
- \textit{Rm} Specifies the second operand register.
- \textit{Ra} Specifies 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**

\[ \text{op}\{\text{cond}\}\{Rd,\} \text{ } Rn, \text{ } Rm \]

where:

- **op** is any of:
  - **USUB16** Unsigned Subtract 16.
  - **USUB8** Unsigned Subtract 8.

- **cond** is an optional condition code, see *Conditional execution on page 3-18*.

- **Rd** specifies the destination register.

- **Rn** specifies the first operand register.

- **Rm** specifies 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 do not change the flags.

**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>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLA</td>
<td>Multiply with Accumulate, 32-bit result</td>
<td>MUL, MLA, and MLS on page 3-75</td>
</tr>
<tr>
<td>MLS</td>
<td>Multiply and Subtract, 32-bit result</td>
<td>MUL, MLA, and MLS on page 3-75</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply, 32-bit result</td>
<td>MUL, MLA, and MLS on page 3-75</td>
</tr>
<tr>
<td>SDIV</td>
<td>Signed Divide</td>
<td>SDIV and UDIV on page 3-94</td>
</tr>
<tr>
<td>SMLAD, SMLADX</td>
<td>Signed Multiply Accumulate Dual</td>
<td>SMLAD on page 3-81</td>
</tr>
<tr>
<td>SMLAL</td>
<td>Signed Multiply with Accumulate (32x32+64), 64-bit result</td>
<td>UMULL, UMLAL, SMULL, and SMLAL on page 3-93</td>
</tr>
<tr>
<td>SMLAL[B,T]</td>
<td>Signed Multiply Accumulate Long (halfwords)</td>
<td>SMLAL and SMLALD on page 3-82</td>
</tr>
<tr>
<td>SMLALD, SMLALDX</td>
<td>Signed Multiply Accumulate Long Dual</td>
<td>SMLAL and SMLALD on page 3-82</td>
</tr>
<tr>
<td>SMLAW[B</td>
<td>T]</td>
<td>Signed Multiply Accumulate (word by halfword)</td>
</tr>
<tr>
<td>SMLSD</td>
<td>Signed Multiply Subtract Dual</td>
<td>SMLSD and SMLSLD on page 3-84</td>
</tr>
<tr>
<td>SMLSDL</td>
<td>Signed Multiply Subtract Long Dual</td>
<td>SMLSD and SMLSLD on page 3-84</td>
</tr>
<tr>
<td>SMMUL</td>
<td>Signed Most Significant Word Multiply Accumulate</td>
<td>SMMLA and SMMLS on page 3-86</td>
</tr>
<tr>
<td>SMMULS, SMMULSR</td>
<td>Signed Most Significant Word Multiply Subtract</td>
<td>SMMLA and SMMLS on page 3-86</td>
</tr>
<tr>
<td>SMUAD, SMUADX</td>
<td>Signed Dual Multiply Add</td>
<td>SMUAD and SMUSD on page 3-89</td>
</tr>
<tr>
<td>SMUL [B,T]</td>
<td>Signed Multiply (word by halfword)</td>
<td>SMUL and SMULW on page 3-91</td>
</tr>
<tr>
<td>SMMUL, SMMULR</td>
<td>Signed Most Significant Word Multiply</td>
<td>SMMUL on page 3-88</td>
</tr>
<tr>
<td>SMULL</td>
<td>Signed Multiply (32x32), 64-bit result</td>
<td>UMULL, UMLAL, SMULL, and SMLAL on page 3-93</td>
</tr>
<tr>
<td>SMULLWB, SMULLWT</td>
<td>Signed Multiply (word by halfword)</td>
<td>SMUL and SMULW on page 3-91</td>
</tr>
<tr>
<td>SMUSD, SMUSDX</td>
<td>Signed Dual Multiply Subtract</td>
<td>SMUAD and SMUSD on page 3-89</td>
</tr>
<tr>
<td>UDIV</td>
<td>Unsigned Divide</td>
<td>SDIV and UDIV on page 3-94</td>
</tr>
<tr>
<td>UMAAL</td>
<td>Unsigned Multiply Accumulate Accumulate Long (32x32+32+32), 64-bit result</td>
<td>UMULL, UMAAL, UMLAL on page 3-77</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned Multiply with Accumulate (32x32+64), 64-bit result</td>
<td>UMULL, UMLAL, SMULL, and SMLAL on page 3-93</td>
</tr>
<tr>
<td>UMLAL</td>
<td>Unsigned Multiply (32x32), 64-bit result</td>
<td>UMULL, UMLAL, SMULL, and SMLAL on page 3-93</td>
</tr>
</tbody>
</table>
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 3-18.
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 3-18.
Rd Specifies 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 $R0$ to $R7$
• $Rd$ must be the same as $Rm$
• you must not use the $cond$ suffix.

Condition flags

If $S$ is specified, the MUL instruction:

• updates the $N$ and $Z$ flags according to the result
• does not affect the $C$ and $V$ flags.

Examples

MUL  R10, R2, R5 ; Multiply, $R10 = R2 \times R5$
MLA  R10, R2, R1, R5 ; Multiply with accumulate, $R10 = (R2 \times R1) + R5$
MULS R0, R2, R2 ; Multiply with flag update, $R0 = R2 \times R2$
MULLT R2, R3, R2 ; Conditionally multiply, $R2 = R3 \times R2$
MLS  R4, R5, R6, R7 ; Multiply with subtract, $R4 = R7 - (R5 \times R6)$
3.6.2 UMULL, UMAAL, UMLAL

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

Syntax

\[ \text{op}\{\text{cond}\} \ RdLo, \ RdHi, \ Rn, \ Rm \]

where:

- **op** is one of:
  - UMULL: Unsigned Long Multiply.
  - UMAAL: Unsigned Long Multiply with Accumulate.
  - UMLAL: Unsigned Long Multiply, with Accumulate.

- **cond** is an optional condition code, see *Conditional execution on page 3-18*.

- **RdHi, RdLo** are the destination registers. For UMAAL, UMLAL and UMLAL they also hold the accumulating value.

- **Rn, Rm** are registers holding the first and second operands.

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 SMLA and SMLAW

Signed Multiply Accumulate (halfwords).

Syntax

op\{XY\}{cond} Rd, Rn, Rm
op\{Y\}{cond} Rd, Rn, Rm, Ra

where:

op  Is one of:
SMLA  Signed Multiply Accumulate Long (halfwords)
X and Y specifies which half of the source registers Rn and Rm are 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.

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

cond  Is an optional condition code, see Conditional execution on page 3-18.
Rd  Specifies 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 SMLBB, SMLABT, SMLATB, SMLATT instructions:
• Multiplies the specified signed halfword, top or bottom, values from Rn and Rm.
• Adds the value in Ra to the resulting 32-bit product.
• Writes the result of the multiplication and addition in Rd.

The non-specified halfwords of the source registers are ignored.

The SMLWB and SMLWBT 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
• Writes 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 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

Signed Multiply Accumulate Long Dual.

Syntax

\[ \text{op}\{X\}\{\text{cond}\} \ Rd, \ Rn, \ Rm, \ Ra ; \]

where:

\text{op} \quad \text{Is one of:}

- \text{SMLAD} \quad \text{Signed Multiply Accumulate Dual}
- \text{SMLADX} \quad \text{Signed Multiply Accumulate Dual Reverse}

\text{X} \quad \text{specifies which halfword of the source register} \ Rn \ \text{is used as the multiply operand. If} \ X \ \text{is omitted, the multiplications are} \bottom \times \bottom \ \text{and top} \times \text{top. If} \ X \ \text{is present, the multiplications are} \bottom \times \text{top and top} \times \text{bottom.}

\text{cond} \quad \text{Is an optional condition code, see Conditional execution on page 3-18.}

\text{Rd} \quad \text{Specifies the destination register.}

\text{Rn} \quad \text{Specifies the first operand register holding the values to be multiplied.}

\text{Rm} \quad \text{Specifies the second operand register.}

\text{Ra} \quad \text{Specifies the accumulate value.}

Operation

The \text{SMLAD} \text{ and} \text{SMLADX} \text{ instructions regard the two operands as four halfword 16-bit values. The} \text{SMLAD} \text{ and} \text{SMLADX} \text{ instructions:}

- \text{If} \ X \ \text{is not present, multiply the top signed halfword value in} \ Rn \ \text{with the top signed halfword of} \ Rm \ \text{and the bottom signed halfword values in} \ Rn \ \text{with the bottom signed halfword of} \ Rm.
- \text{Or if} \ X \ \text{is present, multiply the top signed halfword value in} \ Rn \ \text{with the bottom signed halfword of} \ Rm \ \text{and the bottom signed halfword values in} \ Rn \ \text{with the top signed halfword of} \ Rm.
- \text{Add both multiplication results to the signed 32-bit value in} \ Ra.
- \text{W rites the 32-bit signed result of the multiplication and addition to} \ Rd.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

\begin{verbatim}
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.
\end{verbatim}
3.6.5 SMLAL and SMLALD

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

**Syntax**

\[
\text{op}\{\text{cond}\} \ \text{RdLo}, \ \text{RdHi}, \ \text{Rn}, \ \text{Rm}
\]

\[
\text{op}\{XY\}\{\text{cond}\} \ \text{RdLo}, \ \text{RdHi}, \ \text{Rn}, \ \text{Rm}
\]

\[
\text{op}\{X\}\{\text{cond}\} \ \text{RdLo}, \ \text{RdHi}, \ \text{Rn}, \ \text{Rm}
\]

where:

- \(\text{op}\) is one of:
  - \text{SMLAL} Signed Multiply Accumulate Long
  - \text{SMLAL} Signed Multiply Accumulate Long (halfwords, X and Y)

  X and Y specify which halfword of the source registers \(\text{Rn}\) and \(\text{Rm}\) are used as the first and second multiply operand:

- If \(X = B\), then the bottom halfword, bits [15:0], of \(\text{Rn}\) is used. If \(X = T\), then the top halfword, bits [31:16], of \(\text{Rn}\) is used.

- If \(Y = B\), then the bottom halfword, bits [15:0], of \(\text{Rm}\) is used. If \(Y = T\), then the top halfword, bits [31:16], of \(\text{Rm}\) is used.

- \text{SMLALD} Signed Multiply Accumulate Long Dual
- \text{SMLALDX} Signed Multiply Accumulate Long Dual Reversed

  If the \(X\) is omitted, the multiplications are bottom \(\times\) bottom and top \(\times\) top.

  If \(X\) is present, the multiplications are bottom \(\times\) top and top \(\times\) bottom.

- \(\text{cond}\) is an optional condition code, see *Conditional execution* on page 3-18.

- \(\text{RdHi}, \ \text{RdLo}\) are the destination registers. \(\text{RdLo}\) is the lower 32 bits and \(\text{RdHi}\) is the upper 32 bits of the 64-bit integer. For \text{SMLAL}, \text{SMLALBB}, \text{SMLALBT}, \text{SMLALTB}, \text{SMLALTT}, \text{SMLALD} and \text{SMLALDX}, they also hold the accumulating value.

- \(\text{Rn}, \ \text{Rm}\) are registers holding the first and second operands.

**Operation**

The \text{SMLAL} instruction:

- Multiplies the two’s complement signed word values from \(\text{Rn}\) and \(\text{Rm}\).
- Adds the 64-bit value in \(\text{RdLo}\) and \(\text{RdHi}\) to the resulting 64-bit product.
- Writes the 64-bit result of the multiplication and addition in \(\text{RdLo}\) and \(\text{RdHi}\).

The \text{SMLALBB}, \text{SMLALBT}, \text{SMLALTB} and \text{SMLALTT} instructions:

- Multiplies the specified signed halfword, Top or Bottom, values from \(\text{Rn}\) and \(\text{Rm}\).
- Adds the resulting sign-extended 32-bit product to the 64-bit value in \(\text{RdLo}\) and \(\text{RdHi}\).
- Writes the 64-bit result of the multiplication and addition in \(\text{RdLo}\) and \(\text{RdHi}\).

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:

1. If \( X \) is not present, multiply 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 \( Rm \).
2. Or if \( X \) is present, multiply 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 \( Rm \).
3. Add the two multiplication results to the signed 64-bit value in \( RdLo \) and \( RdHi \) to create the resulting 64-bit product.
4. 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**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMLAL R4, R5, R3, R8</td>
<td>Multiplies R3 and R8, adds R5:R4 and writes to R5:R4</td>
</tr>
<tr>
<td>SMLALBT R2, R1, R6, R7</td>
<td>Multiplies bottom halfword of R6 with top halfword of R7, sign extends to 32-bit, adds R1:R2 and writes to R1:R2</td>
</tr>
<tr>
<td>SMLALTB R2, R1, R6, R7</td>
<td>Multiplies top halfword of R6 with bottom halfword of R7, sign extends to 32-bit, adds R1:R2 and writes to R1:R2</td>
</tr>
<tr>
<td>SMLALD R6, R8, R5, R1</td>
<td>Multiplies top halfwords in R5 and R1 and bottom halfwords of R5 and R1, adds R8:R6 and writes to R8:R6</td>
</tr>
<tr>
<td>SMLALDX R6, R8, R5, R1</td>
<td>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</td>
</tr>
</tbody>
</table>
3.6.6  SMLSD and SMLSLD

Signed Multiply Subtract Dual and Signed Multiply Subtract Long Dual.

Syntax

\[ \text{op}\{X\}\{\text{cond}\} \ R_d, \ R_n, \ R_m, \ R_a \]

where:

\[ \text{op} \]
Is one of:

- \text{SMLSD}  Signed Multiply Subtract Dual.
- \text{SMLSDX}  Signed Multiply Subtract Dual Reversed.
- \text{SMLSLD}  Signed Multiply Subtract Long Dual.
- \text{SMLSLDX}  Signed Multiply Subtract Long Dual Reversed.

If \( X \) is present, the multiplications are bottom \( \times \) top and top \( \times \) bottom. If the \( X \) is omitted, the multiplications are bottom \( \times \) bottom and top \( \times \) top.

\[ \text{cond} \]
Is an optional condition code, see Conditional execution on page 3-18.

\[ R_d \]
Specifies the destination register.

\[ R_n, \ R_m \]
Are registers holding the first and second operands.

\[ R_a \]
Specifies the register holding the accumulate value.

Operation

The \text{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 \( \times \) 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 \text{SMLSLD} instruction interprets the values from \( R_n \) and \( R_m \) as four signed halfwords. This instruction:

- Optionally rotates the halfwords of the second operand.
- Performs two signed 16 \( \times \) 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 \( R_dHi \) and \( R_dLo \) to the result of the subtraction.
- Writes the 64-bit result of the addition to the \( R_dHi \) and \( R_dLo \).

Restrictions

In these instructions:

- Do not use SP and do not use PC.
**Condition flags**

This 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**  
  \[ \text{R0, R4, R5, R6} \]  
  ; 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**  
  \[ \text{R1, R3, R2, R0} \]  
  ; 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**  
  \[ \text{R3, R6, R2, R7} \]  
  ; 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**  
  \[ \text{R3, R6, R2, R7} \]  
  ; 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**

\[
\text{op}\{R\}\{\text{cond}\} \ R_d, \ R_n, \ R_m, \ R_a
\]

where:

- **op** Is one of:
  - \text{SMMLA} Signed Most Significant Word Multiply Accumulate.
  - \text{SMMLS} Signed Most Significant Word Multiply Subtract.

- \text{R} Is a rounding error flag. If \text{R} is specified, the result is rounded instead of being truncated. In this case the constant 0x80000000 is added to the product before the high word is extracted.

- \text{cond} Is an optional condition code, see *Conditional execution* on page 3-18.

- **Rd** Specifies the destination register.

- **Rn, Rm** Are registers holding the first and second multiply operands.

- **Ra** Specifies the register holding the accumulate value.

**Operation**

The **SMMLA** instruction interprets the values from \text{Rn} and \text{Rm} as signed 32-bit words.

The **SMMLA** instruction:
- Multiplies the values in \text{Rn} and \text{Rm}.
- Optionally rounds the result by adding 0x80000000.
- Extracts the most significant 32 bits of the result.
- Adds the value of \text{Ra} to the signed extracted value.
- Writes the result of the addition in \text{Rd}.

The **SMMLS** instruction interprets the values from \text{Rn} and \text{Rm} as signed 32-bit words.

The **SMMLS** instruction:
- Multiplies the values in \text{Rn} and \text{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 \text{Ra}.
- Writes the result of the subtraction in \text{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

\[ \text{op}\{\text{R}\}\{\text{cond}\} \text{ Rd, Rn, Rm} \]

where:

- \( \text{op} \) Is one of:
  - \text{SMMUL} Signed Most Significant Word Multiply
  - \text{R} Is a rounding error flag. If \( \text{R} \) is specified, the result is rounded instead of being truncated. In this case the constant \( 0x80000000 \) is added to the product before the high word is extracted.
  - \( \text{cond} \) Is an optional condition code, see Conditional execution on page 3-18.
  - \( \text{Rd} \) Specifies the destination register.
  - \( \text{Rn}, \text{Rm} \) Are registers holding the first and second operands.

#### Operation

The \text{SMMUL} instruction interprets the values from \( \text{Rn} \) and \( \text{Rm} \) as two’s complement 32-bit signed integers. The \text{SMMUL} instruction:

- Multiplies the values from \( \text{Rn} \) and \( \text{Rm} \).
- Optionally rounds the result, otherwise truncates the result.
- Writes the most significant signed 32 bits of the result in \( \text{Rd} \).

#### 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

- \text{SMULL} \text{ R0, R4, R5} ; Multiplies R4 and R5, truncates top 32 bits and writes to R0
- \text{SMULLR} \text{ 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.

- **cond**
  - Is an optional condition code, see *Conditional execution on page 3-18*.

- **Rd**
  - Specifies 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

Sets 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

\[ \text{op}\{XY\}\{\text{cond}\} \ R_d, \ R_n, \ R_m \]
\[ \text{op}\{Y\}\{\text{cond}\} \ R_d, \ R_n, \ R_m \]

For SMULXY only:

\[ \text{op} \]

Is one of:

\[ \text{SMUL}\{XY\} \]

Signed Multiply (halfwords)

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

\[ \text{SMULW}\{Y\} \]

Signed Multiply (word by halfword)

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

\[ \text{cond} \]

Is an optional condition code, see Conditional execution on page 3-18.

\[ \text{Rd} \]

Specifies the destination register.

\[ \text{Rn}, \ R_m \]

Are registers holding the first and second operands.

Operation

The SMULBB, SMULTB, SMULTT and SMULTT instructions interprets the values from \( R_n \) and \( R_m \) as four signed 16-bit integers. These instructions:

- Multiplies the specified signed halfword, Top or Bottom, values from \( R_n \) and \( R_m \).
- Writes the 32-bit result of the multiplication in \( R_d \).

The SMULWT and SMULWB instructions interprets the values from \( R_n \) as a 32-bit signed integer and \( R_m \) as two halfword 16-bit signed integers. These instructions:

- Multiplies the first operand and the top, T suffix, or the bottom, B suffix, halfword of the second operand.
- Writes 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.
- \( \text{RdHi} \) and \( \text{RdLo} \) 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 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 Long Multiply, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

Syntax

\[ \text{op\{cond\} RdLo, RdHi, Rn, Rm} \]

where:

\( \text{op} \) Is one of:

- \text{UMULL}  Unsigned Long Multiply.
- \text{UMLAL}  Unsigned Long Multiply, with Accumulate.
- \text{SMULL}  Signed Long Multiply.
- \text{SMLAL}  Signed Long Multiply, with Accumulate.

\( \text{cond} \) Is an optional condition code, see Conditional execution on page 3-18.

\( \text{RdHi}, \text{RdLo} \) Are the destination registers. For UMLAL and SMLAL they also hold the accumulating value.

\( \text{Rn}, \text{Rm} \) Are registers holding the operands.

Operation

The \text{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 \text{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 \text{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 \text{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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMULL</td>
<td>R0, R4, R5, R6</td>
<td>( (R4, R5) \times R6 )</td>
</tr>
<tr>
<td>SMLAL</td>
<td>R4, R3, R5, R8</td>
<td>( (R5, R4) \times R3 ) + ( R3 \times R8 )</td>
</tr>
</tbody>
</table>
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 3-18.
Rd Specifies the destination register. If Rd is omitted, the destination register is Rn.
Rn Specifies the register holding the value to be divided.
Rm Is a register holding the divisor.

Operation

SDIV performs a signed integer division of the value in Rn by the value in Rm.
UDIV 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>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSAT</td>
<td>Signed Saturate</td>
<td>SSAT and USAT on page 3-96</td>
</tr>
<tr>
<td>SSAT16</td>
<td>Signed Saturate Halfword</td>
<td>SSAT16 and USAT16 on page 3-97</td>
</tr>
<tr>
<td>USAT</td>
<td>Unsigned Saturate</td>
<td>USAT and SSAT on page 3-96</td>
</tr>
<tr>
<td>USAT16</td>
<td>Unsigned Saturate Halfword</td>
<td>USAT16 and SSAT16 on page 3-97</td>
</tr>
<tr>
<td>QADD</td>
<td>Saturating Add</td>
<td>QADD and QSUB on page 3-98</td>
</tr>
<tr>
<td>QSUB</td>
<td>Saturating Subtract</td>
<td>QADD and QSUB on page 3-98</td>
</tr>
<tr>
<td>QSUB16</td>
<td>Saturating Subtract 16</td>
<td>QADD and QSUB on page 3-98</td>
</tr>
<tr>
<td>QASX</td>
<td>Saturating Add and Subtract with Exchange</td>
<td>QASX and QSAX on page 3-100</td>
</tr>
<tr>
<td>QSAX</td>
<td>Saturating Subtract and Add with Exchange</td>
<td>QASX and QSAX on page 3-100</td>
</tr>
<tr>
<td>QDADD</td>
<td>Saturating Double and Add</td>
<td>QDADD and QDSUB on page 3-102</td>
</tr>
<tr>
<td>QDSUB</td>
<td>Saturating Double and Subtract</td>
<td>QDADD and QDSUB on page 3-102</td>
</tr>
<tr>
<td>UQADD16</td>
<td>Unsigned Saturating Add 16</td>
<td>UQADD and UQSUB on page 3-105</td>
</tr>
<tr>
<td>UQADD8</td>
<td>Unsigned Saturating Add 8</td>
<td>UQADD and UQSUB on page 3-105</td>
</tr>
<tr>
<td>UQASX</td>
<td>Unsigned Saturating Add and Subtract with Exchange</td>
<td>UQASX and UQSAX on page 3-103</td>
</tr>
<tr>
<td>UQSAX</td>
<td>Unsigned Saturating Subtract and Add with Exchange</td>
<td>UQASX and UQSAX on page 3-103</td>
</tr>
<tr>
<td>UQSUB16</td>
<td>Unsigned Saturating Subtract 16</td>
<td>UQADD and UQSUB on page 3-105</td>
</tr>
<tr>
<td>UQSUB8</td>
<td>Unsigned Saturating Subtract 8</td>
<td>UQADD and UQSUB on page 3-105</td>
</tr>
</tbody>
</table>

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 3-164.

To read the state of the Q flag, use the MRS instruction, see MRS on page 3-163.
3.7.1 SSAT and USAT

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

**Syntax**

\[
\text{op}\{\text{cond}\} \text{ Rd, } \#n, \text{ Rm } \{, \text{ shift } \#s\}
\]

where:

- \(\text{op}\) is one of:
  - SSAT: Saturates a signed value to a signed range.
  - USAT: Saturates a signed value to an unsigned range.
- \(\text{cond}\) is an optional condition code, see *Conditional execution* on page 3-18.
- \(\text{Rd}\) specifies 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.
- \(\text{Rm}\) specifies the register containing the value to saturate.
- \(\text{shift } \#s\) is an optional shift applied to \(\text{Rm}\) before saturating. It must be one of the following:
  - ASR \(\#s\) where \(s\) is in the range 1 to 31
  - LSL \(\#s\) where \(s\) is in the range 0 to 31.

**Operation**

These instructions saturate to a signed or unsigned \(n\)-bit value.

The SSAT instruction applies the specified shift, then saturates to the signed range 
\[-2^{n-1} \leq x \leq 2^{n-1}-1.\]

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

**Restrictions**

Do not use SP and do not use PC.

**Condition flags**

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the Q flag to 1.

**Examples**

- SSAT \(\text{R7, } \#16, \text{ R7, LSL } \#4\) ; Logical shift left value in R7 by 4, then
  ; saturate it as a signed 16-bit value and
  ; write it back to R7
- USATNE \(\text{R0, } \#7, \text{ R5}\) ; 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

\[ \text{op}\{\text{cond}\} \text{ Rd, \#n, Rm} \]

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 3-18.

- **Rd** specifies 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** specifies the register containing the value 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

\[
\text{op\{cond\} \{Rd\}, \ Rn, \ Rm} \\
\text{op\{cond\} \{Rd\}, \ Rn, \ Rm}
\]

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 3-18.
- **Rd**
  - Specifies the destination register.
- **Rn, Rm**
  - Are registers holding the first and second operands.

Operation

These instructions add or subtract two, four or eight values from the first and second operands and then writes a signed saturated value in the destination register.

The QADD and QSUB instructions apply the specified add or subtract, and then saturate the result to the signed range \(-2^{n-1} \leq x \leq 2^{n-1}-1\), where \(x\) is given by the number of bits applied in the instruction, 32, 16 or 8.

If the returned result is different from the value to be saturated, it is called saturation. If saturation occurs, the QADD and QSUB instructions set the Q flag to 1 in the APSR. Otherwise, it leaves the Q flag unchanged. The 8-bit and 16-bit QADD and QSUB instructions always leave the Q flag unchanged.

To clear the Q flag to 0, you must use the MSR instruction, see MSR on page 3-164.

To read the state of the Q flag, use the MRS instruction, see MRS on page 3-163.

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**

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

\[
\text{op\{cond\}} \{Rd\}, \ Rm, \ Rn
\]

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 3-18*.
- **Rd** Specifies the destination register.
- **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

QSA  X 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

\[ \text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rm}, \text{Rn} \]

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 3-18.

- **Rd** specifies the destination register.

- **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 \(-2^{31} \leq x \leq 2^{31} - 1\). 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**

\[ \text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rm}, \text{Rn} \]

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 3-18.

- **Rd** Specifies the destination register.

- **Rn, Rm** Are registers holding the first and second operands.

**Operation**

The UQASX instruction:

1. Adds the bottom halfword of the source operand with the top halfword of the second operand.
2. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
3. Saturates the results of the sum and writes a 16-bit unsigned integer in the range \(0 \leq x \leq 2^{16} - 1\), where \(x\) equals 16, to the top halfword of the destination register.
4. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range \(0 \leq x \leq 2^{16} - 1\), where \(x\) equals 16, to the bottom halfword of the destination register.

The UQSAX instruction:

1. Subtracts the bottom halfword of the second operand from the top highword of the first operand.
2. Adds the bottom halfword of the first operand with the top halfword of the second operand.
3. Saturates the result of the subtraction and writes a 16-bit unsigned integer in the range \(0 \leq x \leq 2^{16} - 1\), where \(x\) equals 16, to the top halfword of the destination register.
4. Saturates the results of the addition and writes a 16-bit unsigned integer in the range \(0 \leq x \leq 2^{16} - 1\), 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

UQASX 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**

\[
\text{op}\{\text{cond}\} \{Rd\}, \ Rn, \ Rm \\
\text{op}\{\text{cond}\} \{Rd\}, \ Rn, \ Rm
\]

where:

- \textit{op} is one of:
  - UQADD8: Saturating four unsigned 8-bit integer additions.
  - UQADD16: Saturating two unsigned 16-bit integer additions.
  - UD5UB8: Saturating four unsigned 8-bit integer subtractions.
  - UD5UB16: Saturating two unsigned 16-bit integer subtractions.

- \textit{cond} is an optional condition code, see *Conditional execution* on page 3-18.

- \textit{Rd} specifies the destination register.

- \textit{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 UD5UB16 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 UD5UB8 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 3-11 shows the instructions that operate on packing and unpacking data:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>PKH</td>
<td>Pack Halfword</td>
<td>PKHBT and PKHTB on page 3-108</td>
</tr>
<tr>
<td>SXTAB</td>
<td>Extend 8 bits to 32 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>SXTAB16</td>
<td>Dual extend 8 bits to 16 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>SXTAH</td>
<td>Extend 16 bits to 32 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>SXTB</td>
<td>Sign extend a byte</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>SXTB16</td>
<td>Dual extend 8 bits to 16 and add</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>SXTH</td>
<td>Sign extend a halfword</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>UXTAB</td>
<td>Extend 8 bits to 32 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>UXTAB16</td>
<td>Dual extend 8 bits to 16 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>UXTAH</td>
<td>Extend 16 bits to 32 and add</td>
<td>SXTA and UXTA on page 3-112</td>
</tr>
<tr>
<td>UXTB</td>
<td>Zero extend a byte</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>UXTB16</td>
<td>Dual zero extend 8 bits to 16 and add</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>UXTH</td>
<td>Zero extend a halfword</td>
<td>SXT and UXT on page 3-117</td>
</tr>
</tbody>
</table>
3.8.1 PKHBT and PKHTB

Pack Halfword.

Syntax

\[
\text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rn}, \text{Rm} \{, \text{LSL} \#\text{imm}\}
\]

\[
\text{op}\{\text{cond}\} \{\text{Rd}\}, \text{Rn}, \text{Rm} \{, \text{ASR} \#\text{imm}\}
\]

where:

\text{op} \quad \text{Is one of:}

- PKHBT \quad \text{Pack Halfword, bottom and top with shift.}
- PKHTB \quad \text{Pack Halfword, top and bottom with shift.}

\text{cond} \quad \text{Is an optional condition code, see } \text{Conditional execution on page } 3-18.

\text{Rd} \quad \text{Specifies the destination register.}

\text{Rn} \quad \text{Specifies the first operand register.}

\text{Rm} \quad \text{Specifies the second operand register holding the value to be optionally shifted.}

\text{imm} \quad \text{Specifies the shift length. The type of shift length depends on the instruction:}

For PKHBT:

- LSL \quad \text{A left shift with a shift length from 1 to 31, 0 means no shift.}

For PKHTB:

- ASR \quad \text{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

\text{Rd} \quad \text{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

\[
\text{cond} \quad \{ \text{Rd}, \} \quad \text{Rm} \quad \{, \text{ROR} \ #n\}
\]

where:

- **op**
  - Is one of:
    - **SXTB**: Sign extends an 8-bit value to a 32-bit value.
    - **SXTH**: Sign extends a 16-bit value to a 32-bit value.
    - **SXTB16**: Sign extends two 8-bit values to two 16-bit values.
    - **UXTB**: Zero extends an 8-bit value to a 32-bit value.
    - **UXTH**: Zero extends a 16-bit value to a 32-bit value.
    - **UXTB16**: Zero extends two 8-bit values to two 16-bit values.

- **cond**
  - Is an optional condition code, see Conditional execution on page 3-18.

- **Rd**
  - Specifies the destination register.

- **Rm**
  - Specifies the register holding the value to extend.

- **ROR \ #n**
  - Is one of:
    - **ROR \ #8**: Value from \( \text{Rm} \) is rotated right 8 bits.
    - **ROR \ #16**: Value from \( \text{Rm} \) is rotated right 16 bits.
    - **ROR \ #24**: Value from \( \text{Rm} \) is rotated right 24 bits.
  - If \( \text{ROR} \ #n \) is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from \( \text{Rm} \) 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
; 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

\[ \text{op}\{\text{cond}\} \{\text{Rd},\} \ Rn, \ Rm \ \{, \ \text{ROR} \ #n\} \]

where:

\( \text{op} \) 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.

\( \text{cond} \) Is an optional condition code, see Conditional execution on page 3-18.

\( \text{Rd} \) Specifies the destination register.

\( \text{Rn} \) Specifies the first operand register.

\( \text{Rm} \) Specifies the register holding the value to rotate and extend.

\( \text{ROR} \ #n \) Is one of:
- ROR #8: Value from \( \text{Rm} \) is rotated right 8 bits.
- ROR #16: Value from \( \text{Rm} \) is rotated right 16 bits.
- ROR #24: Value from \( \text{Rm} \) is rotated right 24 bits.

If \( \text{ROR} \ #n \) is omitted, no rotation is performed.

Operation

These instructions do the following:

1. Rotate the value from \( \text{Rm} \) right by 0, 8, 16 or 24 bits.

2. Extract bits from the resulting value:
   - \( \text{SXTAB} \) extracts bits[7:0] from \( \text{Rm} \) and sign extends to 32 bits.
   - \( \text{UXTAB} \) extracts bits[7:0] from \( \text{Rm} \) and zero extends to 32 bits.
   - \( \text{SXTAH} \) extracts bits[15:0] from \( \text{Rm} \) and sign extends to 32 bits.
   - \( \text{UXTAH} \) extracts bits[15:0] from \( \text{Rm} \) and zero extends to 32 bits.
   - \( \text{SXTAB16} \) extracts bits[7:0] from \( \text{Rm} \) and sign extends to 16 bits, and extracts bits [23:16] from \( \text{Rm} \) and sign extends to 16 bits.
   - \( \text{UXTAB16} \) extracts bits[7:0] from \( \text{Rm} \) and zero extends to 16 bits, and extracts bits [23:16] from \( \text{Rm} \) and zero extends to 16 bits.

3. Adds the signed or zero extended value to the word or corresponding halfword of \( \text{Rn} \) and writes the result in \( \text{Rd} \).

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 Bitfield instructions

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

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFC</td>
<td>Bit Field Clear</td>
<td>BFC and BFI on page 3-115</td>
</tr>
<tr>
<td>BFI</td>
<td>Bit Field Insert</td>
<td>BFC and BFI on page 3-115</td>
</tr>
<tr>
<td>SBFX</td>
<td>Signed Bit Field Extract</td>
<td>SBFX and UBFX on page 3-116</td>
</tr>
<tr>
<td>SXTB</td>
<td>Sign extend a byte</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>SXTH</td>
<td>Sign extend a halfword</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>SBFX</td>
<td>Unsigned Bit Field Extract</td>
<td>SBFX and UBFX on page 3-116</td>
</tr>
<tr>
<td>UXTB</td>
<td>Zero extend a byte</td>
<td>SXT and UXT on page 3-117</td>
</tr>
<tr>
<td>UXTH</td>
<td>Zero extend a halfword</td>
<td>SXT and UXT on page 3-117</td>
</tr>
</tbody>
</table>
3.9.1 BFC and BFI

Bit Field Clear and Bit Field Insert.

Syntax

\[
\begin{align*}
\text{BFC}\{\text{cond}\} & \ Rd, \ #\text{lsb}, \ #\text{width} \\
\text{BFI}\{\text{cond}\} & \ Rd, \ Rn, \ #\text{lsb}, \ #\text{width}
\end{align*}
\]

where:

- \text{cond} is an optional condition code, see Conditional execution on page 3-18.
- \text{Rd} specifies the destination register.
- \text{Rn} specifies the source register.
- \text{lsb} specifies the position of the least significant bit of the bitfield. \text{lsb} must be in the range 0 to 31.
- \text{width} specifies the width of the bitfield and must be in the range 1 to 32−\text{lsb}.

Operation

BFC clears a bitfield in a register. It clears \text{width} bits in \text{Rd}, starting at the low bit position \text{lsb}. Other bits in \text{Rd} are unchanged.

BFI copies a bitfield into one register from another register. It replaces \text{width} bits in \text{Rd} starting at the low bit position \text{lsb}, with \text{width} bits from \text{Rn} starting at bit[0]. Other bits in \text{Rd} are unchanged.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

\[
\begin{align*}
\text{BFC} & \ R4, \ #8, \ #12 \quad ; \text{Clear bit 8 to bit 19 (12 bits) of R4 to 0} \\
\text{BFI} & \ R9, \ R2, \ #8, \ #12 \quad ; \text{Replace bit 8 to bit 19 (12 bits) of R9 with bit 0 to bit 11 from R2.}
\end{align*}
\]
3.9.2 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Syntax

\[
\text{SBFX} \{\text{cond}\} \ R_d, \ R_n, \ #\text{lsb}, \ #\text{width} \\
\text{UBFX} \{\text{cond}\} \ R_d, \ R_n, \ #\text{lsb}, \ #\text{width}
\]

where:
\[
\text{cond} \quad \text{Is an optional condition code, see Conditional execution on page 3-18.}
\]
\[
R_d \quad \text{Specifies the destination register.}
\]
\[
R_n \quad \text{Specifies the source register.}
\]
\[
\text{lsb} \quad \text{Specifies the position of the least significant bit of the bitfield. lsb must be in the range 0 to 31.}
\]
\[
\text{width} \quad \text{Specifies the width of the bitfield and must be in the range 1 to 32−lsb.}
\]

Operation

SBFX extracts a bitfield from one register, sign extends it to 32 bits, and writes the result to the destination register.

UBFX extracts a bitfield 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

\[
\text{SBFX} \ R_0, \ R_1, \ #20, \ #4 \quad ; \text{Extract bit 20 to bit 23 (4 bits) from R1 and sign extend to 32 bits and then write the result to R0.}
\]
\[
\text{UBFX} \ R_8, \ R_{11}, \ #9, \ #10 \quad ; \text{Extract bit 9 to bit 18 (10 bits) from R11 and zero extend to 32 bits and then write the result to R8.}
\]
3.9.3 SXT and UXT

Sign extend and Zero extend.

**Syntax**

SXT\{cond\} \{Rd,\} Rm {, ROR \#n}

UXT\{cond\} \{Rd\}, Rm {, 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 3-18.

- **Rd** Specifies the destination register.

- **Rm** Specifies 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.

**Restrictions**

Do not use SP and do not use PC.

**Condition flags**

These instructions do not affect the flags.

**Examples**

```
SXTB  R3, R10 ; Extract lowest byte of the value in R10 and zero extend it, and write the result to R3.
```

```
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.
```
3.10 Branch and control instructions

Table 3-13 shows the branch and control instructions:

Table 3-13 Branch and control instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Branch</td>
<td><em>B, BL, BX, and BLX on page 3-119</em></td>
</tr>
<tr>
<td>BL</td>
<td>Branch with Link</td>
<td><em>B, BL, BX, and BLX on page 3-119</em></td>
</tr>
<tr>
<td>BLX</td>
<td>Branch indirect with Link</td>
<td><em>B, BL, BX, and BLX on page 3-119</em></td>
</tr>
<tr>
<td>BX</td>
<td>Branch indirect</td>
<td><em>B, BL, BX, and BLX on page 3-119</em></td>
</tr>
<tr>
<td>CBNZ</td>
<td>Compare and Branch if Non Zero</td>
<td><em>CBZ and CBNZ on page 3-121</em></td>
</tr>
<tr>
<td>CBZ</td>
<td>Compare and Branch if Zero</td>
<td><em>CBZ and CBNZ on page 3-121</em></td>
</tr>
<tr>
<td>IT</td>
<td>If-Then</td>
<td><em>IT on page 3-122</em></td>
</tr>
<tr>
<td>TBB</td>
<td>Table Branch Byte</td>
<td><em>TBB and TBH on page 3-124</em></td>
</tr>
<tr>
<td>TBH</td>
<td>Table Branch Halfword</td>
<td><em>TBB and TBH on page 3-124</em></td>
</tr>
</tbody>
</table>
3.10.1 B, BL, BX, and BLX

Branch instructions.

**Syntax**

\[
\begin{align*}
&B\{\text{cond}\} \text{ label} \\
&BL\{\text{cond}\} \text{ label} \\
&BX\{\text{cond}\} Rm \\
&BLX\{\text{cond}\} Rm \\
\end{align*}
\]

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 3-18*.

**label** is a PC-relative expression. See *PC-relative expressions on page 3-17*.

**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 3-122*.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Branch range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B label</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>B{cond} label (outside IT block)</td>
<td>−1 MB to +1 MB</td>
</tr>
<tr>
<td>B{cond} label (inside IT block)</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>BL{cond} label</td>
<td>−16 MB to +16 MB</td>
</tr>
<tr>
<td>BX{cond} Rm</td>
<td>Any value in register</td>
</tr>
<tr>
<td>BLX{cond} Rm</td>
<td>Any value in register</td>
</tr>
</tbody>
</table>
Note

You might have to use the .W suffix to get the maximum branch range. See Instruction width selection on page 3-21.

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

Bcond 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**

CBZ $Rn$, $label$

CBNZ $Rn$, $label$

where:

$Rn$ Specifies the register holding the operand.

$label$ Specifies 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 $R0$ to $R7$
- 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 Bcond must be inside an IT block. Bcond can be either
  outside or inside an IT block but has a larger branch range if it is inside one

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

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

Condition flags

This instruction does not change the flags.

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
ADDCT R1, R0, #55 ; Convert 0xA -> 'A'
ADDLE R1, R0, #48 ; Convert 0x0 -> '0'

IT  GT          ; IT block with only one conditional instruction
ADDCT 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 Specifies 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 Specifies 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 twice 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

BranchTable.H
DCI ((CaseA - BranchTable.H)/2) ; CaseA offset calculation
DCI ((CaseB - BranchTable.H)/2) ; CaseB offset calculation
DCI ((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

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 4-52 for information about enabling the floating-point unit.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VABS</td>
<td>Floating-point Absolute</td>
<td>VABS on page 3-128</td>
</tr>
<tr>
<td>VADD</td>
<td>Floating-point Add</td>
<td>VADD on page 3-129</td>
</tr>
<tr>
<td>VCMP</td>
<td>Compare two floating-point registers, or one floating-point register and zero</td>
<td>VCMP, VCMPE on page 3-130</td>
</tr>
<tr>
<td>VCMPE</td>
<td>Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check</td>
<td>VCMP, VCMPE on page 3-130</td>
</tr>
<tr>
<td>VCVT</td>
<td>Convert between floating-point and integer</td>
<td>VCVT, VCVTR between floating-point and integer on page 3-131</td>
</tr>
<tr>
<td>VCVT</td>
<td>Convert between floating-point and fixed point</td>
<td>VCVT between floating-point and fixed-point on page 3-132</td>
</tr>
<tr>
<td>VCVTR</td>
<td>Convert between floating-point and integer with rounding</td>
<td>VCVT, VCVTR between floating-point and integer on page 3-131</td>
</tr>
<tr>
<td>VCVTB</td>
<td>Converts half-precision value to single-precision</td>
<td>VCVTB, VCVTT on page 3-133</td>
</tr>
<tr>
<td>VCVTT</td>
<td>Converts single-precision register to half-precision</td>
<td>VCVTB, VCVTT on page 3-133</td>
</tr>
<tr>
<td>VDIV</td>
<td>Floating-point Divide</td>
<td>VDIV on page 3-134</td>
</tr>
<tr>
<td>VFMA</td>
<td>Floating-point Fused Multiply Accumulate</td>
<td>VFMA, VFMS on page 3-135</td>
</tr>
<tr>
<td>VFNMA</td>
<td>Floating-point Fused Negate Multiply Accumulate</td>
<td>VFNMA, VFNMS on page 3-136</td>
</tr>
<tr>
<td>VFMS</td>
<td>Floating-point Fused Multiply Subtract</td>
<td>VFMA, VFMS on page 3-135</td>
</tr>
<tr>
<td>VFNMS</td>
<td>Floating-point Fused Negate Multiply Subtract</td>
<td>VFNMA, VFNMS on page 3-136</td>
</tr>
<tr>
<td>VLDM</td>
<td>Load Multiple extension registers</td>
<td>VLDM on page 3-137</td>
</tr>
<tr>
<td>VLDR</td>
<td>Loads an extension register from memory</td>
<td>VLDR on page 3-138</td>
</tr>
<tr>
<td>VLMA</td>
<td>Floating-point Multiply Accumulate</td>
<td>VLMA, VLMS on page 3-139</td>
</tr>
<tr>
<td>VLMS</td>
<td>Floating-point Multiply Subtract</td>
<td>VLMA, VLMS on page 3-139</td>
</tr>
<tr>
<td>VM0V</td>
<td>Floating-point Move Immediate</td>
<td>VMOV Immediate on page 3-140</td>
</tr>
<tr>
<td>VM0V</td>
<td>Floating-point Move Register</td>
<td>VMOV Register on page 3-141</td>
</tr>
<tr>
<td>VM0V</td>
<td>Copy ARM core register to single precision</td>
<td>VMOV ARM Core register to single precision on page 3-143</td>
</tr>
<tr>
<td>VM0V</td>
<td>Copy 2 ARM core registers to 2 single precision</td>
<td>VMOV Two ARM Core registers to two single precision on page 3-144</td>
</tr>
</tbody>
</table>
Table 3-15 Floating-point instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMOV</td>
<td>Copies between ARM core register to scalar</td>
<td>VMOV ARM Core register to scalar on page 3-145</td>
</tr>
<tr>
<td>VMOV</td>
<td>Copies between Scalar to ARM core register</td>
<td>VMOV Scalar to ARM Core register on page 3-142</td>
</tr>
<tr>
<td>VMRS</td>
<td>Move to ARM core register from floating-point System Register</td>
<td>VMRS on page 3-146</td>
</tr>
<tr>
<td>VMRS</td>
<td>Move to floating-point System Register from ARM Core register</td>
<td>VMRS on page 3-147</td>
</tr>
<tr>
<td>VMUL</td>
<td>Multiply floating-point</td>
<td>VMUL on page 3-148</td>
</tr>
<tr>
<td>VNEG</td>
<td>Floating-point negate</td>
<td>VNEG on page 3-149</td>
</tr>
<tr>
<td>VNMLA</td>
<td>Floating-point multiply and add</td>
<td>VNMLA, VNMLS, VNMUL on page 3-150</td>
</tr>
<tr>
<td>VNMLS</td>
<td>Floating-point multiply and subtract</td>
<td>VNMLA, VNMLS, VNMUL on page 3-150</td>
</tr>
<tr>
<td>VNMUL</td>
<td>Floating-point multiply</td>
<td>VNMLA, VNMLS, VNMUL on page 3-150</td>
</tr>
<tr>
<td>VPOP</td>
<td>Pop extension registers</td>
<td>VPOP on page 3-151</td>
</tr>
<tr>
<td>VPUSH</td>
<td>Push extension registers</td>
<td>VPUSH on page 3-152</td>
</tr>
<tr>
<td>VSQRT</td>
<td>Floating-point square root</td>
<td>VSQRT on page 3-153</td>
</tr>
<tr>
<td>VSTM</td>
<td>Store Multiple extension registers</td>
<td>VSTM on page 3-154</td>
</tr>
<tr>
<td>VSTR</td>
<td>Stores an extension register to memory</td>
<td>VSTR on page 3-155</td>
</tr>
<tr>
<td>VSUB</td>
<td>Floating-point Subtract</td>
<td>VSUB on page 3-156</td>
</tr>
</tbody>
</table>
3.11.1 VABS

Floating-point Absolute.

Syntax

VABS{cond}.F32 Sd, Sm

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

Sd, Sm Are the destination floating-point value and the operand floating-point value.

Operation

This instruction:

1. Takes the absolute value of the operand floating-point register.
2. Places the results in the destination floating-point register.

Restrictions

There are no restrictions.

Condition flags

The floating-point instruction clears the sign bit.

Examples

VABS.F32 S4, S6
3.11.2 VADD

Floating-point Add.

**Syntax**

\[
\text{VADD}\{\text{cond}\}.F32 \{Sd,\} \ Sn, Sm
\]

where:
- **cond** Is an optional condition code, see *Conditional execution on page 3-18*.
- **Sd** Specifies the destination floating-point value.
- **Sn, Sm** Are the operand floating-point values.

**Operation**

This instruction:
1. Adds the values in the two floating-point operand registers.
2. Places the results in the destination floating-point register.

**Restrictions**

There are no restrictions.

**Condition flags**

This instruction does not change the flags.

**Examples**

\[
\text{VADD}.F32 S4, S6, S7
\]
3.11.3 VCMP, VCMPE

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

**Syntax**

VCMP{E}{cond}.F32  Sd,  Sm
VCMP{E}{cond}.F32  Sd,  #0.0

where:
- **cond** is an optional condition code, see *Conditional execution on page 3-18*.
- **E** If present, any NaN operand causes an Invalid Operation exception. Otherwise, only a signaling NaN causes the exception.
- **Sd** Specifies the floating-point operand to compare.
- **Sm** Specifies the floating-point operand that is compared with.

**Operation**

This instruction:

1. Compares:
   - Two floating-point registers.
   - 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 3-146*.

**Examples**

VCMP.F32  S4,  #0.0
VCMP.F32  S4,  S2
3.11.4 VCVT, VCVTR between floating-point and integer

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

Syntax

VCVT{R}{cond} . F32 Sd, Sm
VCVT{cond} . F32 Tm Sd, Sm

where:

- \( R \) If \( R \) is specified, the operation uses the rounding mode specified by the FPSCR. If \( R \) is omitted, the operation uses the Round towards Zero rounding mode.
- \( cond \) Is an optional condition code, see Conditional execution on page 3-18.
- \( Tm \) Specifies the data type for the operand. It must be one of:
  - S32 signed 32-bit value.
  - U32 unsigned 32-bit value.
- \( Sd, Sm \) Are the destination register and the operand register.

Operation

These instructions:

1. Either
   - Converts a value in a register from floating-point value to a 32-bit integer.
   - Converts from a 32-bit integer to floating-point value.
2. Places 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.F32 Sd, Sd, #fbits
VCVT{cond}.F32.Td Sd, Sd, #fbits

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Td Specifies 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 Specifies the destination register and the operand register.
fbits Specifies 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

These instructions:
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.

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, VCVTT

Converts between a half-precision value and a single-precision value.

Syntax

VCVT{y}{cond}.F32.F16  Sd,  Sm
VCVT{y}{cond}.F16.F32  Sd,  Sm

where:
y Specifies which half of the operand register Sm or destination register Sd is used for the operand or destination:
  • If y is B, then the bottom half, bits [15:0], of Sm or Sd is used.
  • If y is T, then the top half, bits [31:16], of Sm or Sd is used.
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination register.
Sm Specifies the operand register.

Operation

This instruction with the .F16.32 suffix:
1. Converts the half-precision value in the top or bottom half of a single-precision. register to single-precision.
2. Writes the result to a single-precision register.

This instruction with the .F32.F16 suffix:
1. Converts the value in a single-precision register to half-precision.
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}.F32 {Sd,} S\(n\), Sm

where:

- **cond** Is an optional condition code, see *Conditional execution on page 3-18*.
- **Sd** Specifies the destination register.
- **Sn, Sm** Are the operand registers.

**Operation**

This instruction:
1. Divides one floating-point value by another floating-point value.
2. Writes the result to the floating-point destination register.

**Restrictions**

There are no restrictions.

**Condition flags**

These instructions do not change the flags.
3.11.8 VFMA, VFMS

Floating-point Fused Multiply Accumulate and Subtract.

Syntax

VFMA{cond}.F32 {Sd,} Sn, Sm
VFMS{cond}.F32 {Sd,} Sn, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination register.
Sn, Sm Are the operand registers.

Operation

The VFMA instruction:
1. Multiplies the floating-point values in the operand registers.
2. Accumulates the results into the destination register.

The result of the multiply is not rounded before the accumulation.

The VFMS instruction:
1. Negates the first operand register.
2. Multiplies the floating-point values of the first and second operand registers.
3. Adds the products to the destination register.
4. Places the results in the destination register.

The result of the multiply is not rounded before the addition.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.
3.11.9 VFNMA, VFNMS

Floating-point Fused Negate Multiply Accumulate and Subtract.

Syntax

VFNMA{cond}.F32 {Sd,} Sn, Sm
VFNMS{cond}.F32 {Sd,} Sn, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination register.
Sn, Sm Are the operand registers.

Operation

The VFNMA instruction:
1. Negates the first floating-point operand register.
2. Multiplies the first floating-point operand with second floating-point operand.
3. Adds the negation of the floating-point destination register to the product
4. Places the result into the destination register.

The result of the multiply is not rounded before the addition.

The VFNMS instruction:
1. Multiplies the first floating-point operand with second floating-point operand.
2. Adds the negation of the floating-point value in the destination register to the product.
3. Places the result in the destination register.

The result of the multiply is not rounded before the addition.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.
3.11.10 VLDM

Floating-point Load Multiple.

Syntax

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

where:

mode Specifies the addressing mode:
- IA Increment After. The consecutive addresses start at the address specified in Rn.
- 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 3-18.

size Is an optional data size specifier.

Rn Specifies 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 Specifies 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.
3.11.11 VLDR

Loads a single extension register from memory.

Syntax

VLDR\{cond\}\{.64\} \(Dd, [Rn\#imm]\)
VLDR\{cond\}\{.64\} \(Dd, label\)
VLDR\{cond\}\{.64\} \(Dd, [PC, \#imm]\)
VLDR\{cond\}\{.32\} \(Sd, [Rn\{, \#imm}\]
VLDR\{cond\}\{.32\} \(Sd, label\)
VLDR\{cond\}\{.32\} \(Sd, [PC, \#imm]\)

where:

- **cond** Is an optional condition code, see Conditional execution on page 3-18.
- **64, 32** Are the optional data size specifiers.
- **Dd** Specifies the destination register for a doubleword load.
- **Sd** Specifies the destination register for a singleword load.
- **Rn** Specifies 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 to 1020.
- **label** Specifies 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 VLMA, VLMS

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

Syntax

VLMA{cond}.F32  Sd,  Sn,  Sm
VLMS{cond}.F32  Sd,  Sn,  Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination floating-point value.
Sn, Sm Are the operand floating-point values.

Operation

The floating-point Multiply Accumulate instruction:
1. Multiplies two floating-point values.
2. Adds the results to the destination floating-point value.

The floating-point Multiply Subtract instruction:
1. Multiplies two floating-point values.
2. Subtracts the products from the destination floating-point value.
3. Places the results in the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.
3.11.13 VMOV Immediate

Move floating-point Immediate.

Syntax

\[ \text{VMOV}\{\text{cond}\}.\text{F32} \ Sd, \#\text{imm} \]

where:

- \text{cond} is an optional condition code, see \textit{Conditional execution on page 3-18.}
- \text{Sd} specifies the branch destination.
- \text{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}.F64  Dd,  Dm
VMOV{cond}.F32  Sd,  Sm

where:

cond    Is an optional condition code, see Conditional execution on page 3-18.
Dd      Specifies the destination register, for a doubleword operation.
Dm      Specifies the source register, for a doubleword operation.
Sd      Specifies the destination register, for a singleword operation.
Sm      Specifies 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

\[ \text{VMOV} \{\text{cond}\} \ Rt, \ Dn[x] \]

where:

- \text{cond} is an optional condition code, see \textit{Conditional execution on page 3-18}.
- \text{Rt} specifies the destination ARM core register.
- \text{Dn} specifies the 64-bit doubleword register.
- \text{x} specifies which half of the doubleword register to use:
  - If \text{x} is 0, use lower half of doubleword register
  - If \text{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

\text{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

\[
\begin{align*}
\text{VMOV}\{\text{cond}\} & \ Sn, \ Rt \\
\text{VMOV}\{\text{cond}\} & \ Rt, \ Sn \\
\end{align*}
\]

where:

\begin{align*}
\text{cond} & \quad \text{Is an optional condition code, see Conditional execution on page 3-18.} \\
Sn & \quad \text{Specifies the single-precision floating-point register.} \\
Rt & \quad \text{Specifies the ARM core register.}
\end{align*}

Operation

This instruction transfers:

\begin{itemize}
  \item The contents of a single-precision register to an ARM core register.
  \item The contents of an ARM core register to a single-precision register.
\end{itemize}

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

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

**Syntax**

\[
\text{VMOV\{cond\} } Sm, Sm1, Rt, Rt2 \\
\text{VMOV\{cond\} } Rt, Rt2, Sm, Sm
\]

where:

- \text{cond} is an optional condition code, see *Conditional execution on page 3-18*.
- \text{Sm} specifies the first single-precision register.
- \text{Sm1} specifies the second single-precision register. This is the next single-precision register after \text{Sm}.
- \text{Rt} specifies the ARM core register that \text{Sm} is transferred to or from.
- \text{Rt2} specifies the ARM core register that \text{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.
- \text{Rt} cannot be PC or SP.

**Condition flags**

These instructions do not change the flags.
3.11.18 VMOV ARM Core register to scalar

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

**Syntax**

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

where:

- \( \text{cond} \) is an optional condition code, see *Conditional execution on page 3-18*.
- \( .32 \) is an optional data size specifier.
- \( Dd[x] \) specifies 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.
- \( Rt \) specifies 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.
3.11.19 VMRS

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

Syntax

\texttt{VMRS\{cond\} \ Rt, \ FPSCR}

\texttt{VMRS\{cond\} \ APSR\_nzcv, \ FPSCR}

where:

\texttt{cond} Is an optional condition code, see \textit{Conditional execution on page 3-18}.

\texttt{Rt} Specifies the destination ARM core register. This register can be R0-R14.

\texttt{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

\texttt{Rt} cannot be PC or SP.

Condition flags

These instructions optionally change the flags: N, Z, C, V
3.11.20  VMSR

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

Syntax

VMSR{cond} FPSCR, R
t

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Rt Specifies 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 4-50 for more information.

Restrictions

The restrictions are:
• Rt cannot be PC or SP.

Condition flags

This instruction updates the FPSCR.
3.11.21 VMUL

Floating-point Multiply.

Syntax

```
VMUL{cond}.F32 {Sd,} Sn, Sm
```

where:

- `cond` Is an optional condition code, see *Conditional execution on page 3-18*.
- `Sd` Specifies the destination floating-point value.
- `Sn, Sm` Are the operand floating-point values.

Operation

This instruction:

1. Multiplies two floating-point values.
2. Places the results in the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.
3.11.22 VNEG

Floating-point Negate.

Syntax

VNEG{cond}.F32 Sd, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination floating-point value.
Sm Specifies 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.
3.11.23 VNMLA, VNMLS, VNMUL

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

Syntax

VNMLA{cond}.F32 Sd, Sn, Sm
VNMLS{cond}.F32 Sd, Sn, Sm
VNMUL{cond}.F32 {Sd,} Sn, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination floating-point register.
Sn, Sm Are the operand floating-point registers.

Operation

The VNMLA instruction:
1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the negation of the product.
3. Writes the result back to the destination register.

The VNMLS instruction:
1. Multiplies two floating-point register values.
2. Adds the negation of the floating-point value in the destination register to the product.
3. Writes the result back to the destination register.

The VNMUL instruction:
1. Multiplies together two floating-point register values.
2. Writes the negation of the result to the destination register.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.
3.11.24 VPOP

Floating-point extension register Pop.

**Syntax**

VPOP{cond}{.size} list

where:

- **cond** Is an optional condition code, see *Conditional execution on page 3-18*.
- **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.
3.11.25 VPUSH

Floating-point extension register Push.

Syntax

VPUSH{cond}{.size} list

where:
cond  Is an optional condition code, see Conditional execution on page 3-18.
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

The restrictions are:

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

Condition flags

These instructions do not change the flags.
3.11.26 VSQRT

Floating-point Square Root.

Syntax

VSQRT{cond}.F32 Sd, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination floating-point value.
Sm Specifies 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.
3.11.27 VSTM

Floating-point Store Multiple.

Syntax

\[
\text{VSTM}\{\text{mode}\}\{\text{cond}\}\{.\text{size}\} \quad \text{Rn}\{!\}, \text{list}
\]

where:

- **mode**: Specifies the addressing mode:
  - IA *Increment After*. The consecutive addresses start at the address specified in \( \text{Rn} \). This is the default and can be omitted.
  - DB *Decrement Before*. The consecutive addresses end just before the address specified in \( \text{Rn} \).

- **cond**: Is an optional condition code, see *Conditional execution on page 3-18*.

- **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 \( \text{list} \).

- **Rn**: Specifies the base register. The SP can be used.

- **!**: is the function that causes the instruction to write a modified value back to \( \text{Rn} \).
  - Required if \( \text{mode} = \text{DB} \).

- **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:

- \( \text{list} \) must contain at least one register. If it contains doubleword registers it must not contain more than 16 registers.
- Use of the PC as \( \text{Rn} \) is deprecated.

Condition flags

These instructions do not change the flags.
3.11.28 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 3-18.
32, 64 Are the optional data size specifiers.
Sd Specifies the source register for a singleword store.
Dd Specifies the source register for a doubleword store.
Rn Specifies 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 restrictions are:

• The use of PC for Rn is deprecated.

Condition flags

These instructions do not change the flags.
3.11.29 VSUB

Floating-point Subtract.

Syntax

VSUB{cond}.F32 {Sd,} Sn, Sm

where:
cond Is an optional condition code, see Conditional execution on page 3-18.
Sd Specifies the destination floating-point value.
Sn, Sm Are the operand floating-point value.

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.
3.12 Miscellaneous instructions

Table 3-16 shows the remaining Cortex-M4 instructions:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Brief description</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>BKPT</td>
<td>Breakpoint</td>
<td>BKPT on page 3-158</td>
</tr>
<tr>
<td>CPSID</td>
<td>Change Processor State, Disable Interrupts</td>
<td>CPS on page 3-159</td>
</tr>
<tr>
<td>CPSIE</td>
<td>Change Processor State, Enable Interrupts</td>
<td>CPS on page 3-159</td>
</tr>
<tr>
<td>DMB</td>
<td>Data Memory Barrier</td>
<td>DMB on page 3-160</td>
</tr>
<tr>
<td>DSB</td>
<td>Data Synchronization Barrier</td>
<td>DSB on page 3-161</td>
</tr>
<tr>
<td>ISB</td>
<td>Instruction Synchronization Barrier</td>
<td>ISB on page 3-162</td>
</tr>
<tr>
<td>MRS</td>
<td>Move from special register to register</td>
<td>MRS on page 3-163</td>
</tr>
<tr>
<td>MSR</td>
<td>Move from register to special register</td>
<td>MSR on page 3-164</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
<td>NOP on page 3-165</td>
</tr>
<tr>
<td>SEV</td>
<td>Send Event</td>
<td>SEV on page 3-166</td>
</tr>
<tr>
<td>SVC</td>
<td>Supervisor Call</td>
<td>SVC on page 3-167</td>
</tr>
<tr>
<td>WFE</td>
<td>Wait For Event</td>
<td>WFE on page 3-168</td>
</tr>
<tr>
<td>WFI</td>
<td>Wait For Interrupt</td>
<td>WFI on page 3-169</td>
</tr>
</tbody>
</table>
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

```
CPS<effect> <iflags>
```

where:

- **effect** is one of:
  - IE: Clears the special purpose register.
  - ID: Sets the special purpose register.

- **iflags** is a sequence of one or more flags:
  - i: Set or clear PRIMASK.
  - f: Set or clear FAULTMASK.

#### Operation

CPS changes the PRIMASK and FAULTMASK special register values. 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 DMB

Data Memory Barrier.

Syntax

DMB{cond};

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

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
3.12.4 DSB

Data Synchronization Barrier.

Syntax

DSB{cond}

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

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
3.12.5 ISB

Instruction Synchronization Barrier.

Syntax

ISB{cond}

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

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
3.12.6 MRS

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

Syntax

\[ \text{MRS}\{\text{cond}\} \, \text{Rd}, \, \text{spec}\_\text{reg} \]

where:

- \text{cond} is an optional condition code, see \textit{Conditional execution on page 3-18}.
- \text{Rd} specifies the destination register.
- \text{spec}\_\text{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 3-164.

Restrictions

\text{Rd} must not be SP and must not be PC.

Condition flags

This instruction does not change the flags.

Examples

\[ \text{MRS} \, \text{R0}, \, \text{PRIMASK} \, ; \, \text{Read PRIMASK value and write it to R0} \]
3.12.7 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 3-18.
Rn Specifies 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 3-163.

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
### 3.12.8 NOP

No Operation.

**Syntax**

NOP{cond};

where:

- **cond** is an optional condition code, see *Conditional execution on page 3-18*.

**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
3.12.9 SEV

Send Event.

Syntax

SEV{cond};

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

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 2-32.

Condition flags

This instruction does not change the flags.

Examples

SEV ; Send Event
3.12.10 SVC

Supervisor Call.

Syntax

\[ \text{SVC}\{\text{cond}\} \ #\text{imm} \]

where:

\text{cond} \quad \text{Is an optional condition code, see Conditional execution on page 3-18.}

\text{imm} \quad \text{Is an expression evaluating to an integer in the range 0-255 (8-bit value).}

Operation

The SVC instruction causes the SVC exception.

\text{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

\[ \text{SVC} \ #0x32 \ ; \ \text{Supervisor Call} \ (\text{SVCall} \ \text{handler} \ \text{can} \ \text{extract} \ \text{the} \ \text{immediate} \ \text{value} \]
\[ \quad \text{by} \ \text{locating} \ \text{it} \ \text{through} \ \text{the} \ \text{stacked} \ \text{PC} \]
3.12.11 WFE

Wait For Event.

Syntax

WFE{cond};

where:

cond Is an optional condition code, see Conditional execution on page 3-18.

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 2-32.

Condition flags

This instruction does not change the flags.

Examples

        WFE ; Wait for event
3.12.12 WFI

Wait for Interrupt.

**Syntax**

WFI\{cond\}

where:

\(cond\) Is an optional condition code, see *Conditional execution* on page 3-18.

**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-M4 Peripherals

This chapter describes the ARM Cortex-M4 core peripherals. It contains the following sections:

- *About the Cortex-M4 peripherals* on page 4-2
- *Nested Vectored Interrupt Controller* on page 4-3
- *System control block* on page 4-11
- *System timer, SysTick* on page 4-33.
- *Optional Memory Protection Unit* on page 4-37
- *Floating Point Unit (FPU)* on page 4-48.
4.1 About the Cortex-M4 peripherals

The address map of the *Private Peripheral Bus* (PPB) is:

<table>
<thead>
<tr>
<th>Address</th>
<th>Core peripheral</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E008-0xE000E00F</td>
<td>System Control Block</td>
<td>Table 4-12 on page 4-11</td>
</tr>
<tr>
<td>0xE000E010-0xE000E01F</td>
<td>System timer</td>
<td>Table 4-32 on page 4-33</td>
</tr>
<tr>
<td>0xE000E100-0xE000E4EF</td>
<td>Nested Vectored Interrupt Controller</td>
<td>Table 4-2 on page 4-3</td>
</tr>
<tr>
<td>0xE000E000-0xE000E3F</td>
<td>System Control Block</td>
<td>Table 4-12 on page 4-11</td>
</tr>
<tr>
<td>0xE000ED90-0xE000ED93</td>
<td>MPU Type Register</td>
<td>Reads as zero, indicating MPU is not implemented&lt;sup&gt;a&lt;/sup&gt;</td>
</tr>
<tr>
<td>0xE000ED90-0xE000EDB8</td>
<td>Memory Protection Unit</td>
<td>Table 4-38 on page 4-38</td>
</tr>
<tr>
<td>0xE000EF00-0xE000EF03</td>
<td>Nested Vectored Interrupt Controller</td>
<td>Table 4-2 on page 4-3</td>
</tr>
<tr>
<td>0xE000EF30-0xE000EF44</td>
<td>Floating Point Unit</td>
<td>Table 4-49 on page 4-48</td>
</tr>
</tbody>
</table>

<sup>a</sup> 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.
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 interrupts.
- A 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>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Required privilege</th>
<th>Reset value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E100-0xE000E11C</td>
<td>NVIC_ISER0-NVIC_ISER7</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Set-enable Registers on page 4-4</td>
</tr>
<tr>
<td>0xE000E180-0xE000E19C</td>
<td>NVIC_ICER0-NVIC_ICER7</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Clear-enable Registers on page 4-5</td>
</tr>
<tr>
<td>0xE000E200-0xE000E21C</td>
<td>NVIC_ISPR0-NVIC_ISPR7</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Set-pending Registers on page 4-5</td>
</tr>
<tr>
<td>0xE000E280-0xE000E29C</td>
<td>NVIC_ICPR0-NVIC_ICPR7</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Clear-pending Registers on page 4-6</td>
</tr>
<tr>
<td>0xE000E300-0xE000E31C</td>
<td>NVIC_IABR0-NVIC_IABR7</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Active Bit Registers on page 4-7</td>
</tr>
<tr>
<td>0xE000E400-0xE000E4EF</td>
<td>NVIC_IPR0-NVIC_IPR59</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Priority Registers on page 4-7</td>
</tr>
<tr>
<td>0xE000EF00</td>
<td>STIR</td>
<td>WO</td>
<td>Configurable(^a)</td>
<td>0x00000000</td>
<td>Software Trigger Interrupt Register on page 4-8</td>
</tr>
</tbody>
</table>

\(^a\) See the register description for more information.
4.2.1 Accessing the Cortex-M4 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

<table>
<thead>
<tr>
<th>CMSIS function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void NVIC_EnableIRQ(IRQn_Type IRQn)</td>
<td>Enables an interrupt or exception.</td>
</tr>
<tr>
<td>void NVIC_DisableIRQ(IRQn_Type IRQn)</td>
<td>Disables an interrupt or exception.</td>
</tr>
<tr>
<td>void NVIC_SetPendingIRQ(IRQn_Type IRQn)</td>
<td>Sets the pending status of interrupt or exception to 1.</td>
</tr>
<tr>
<td>void NVIC_ClearPendingIRQ(IRQn_Type IRQn)</td>
<td>Clears the pending status of interrupt or exception to 0.</td>
</tr>
<tr>
<td>uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)</td>
<td>Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.</td>
</tr>
<tr>
<td>void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)</td>
<td>Sets the priority of an interrupt or exception with configurable priority level to 1.</td>
</tr>
<tr>
<td>uint32_t NVIC_GetPriority(IRQn_Type IRQn)</td>
<td>Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.</td>
</tr>
</tbody>
</table>

a. The input parameter IRQn is the IRQ number, see Table 2-16 on page 2-22 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:

![Set Enable Bits](image)

Table 4-4 ISER bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>SETENA</td>
<td>Interrupt set-enable bits. Write: 0 = no effect 1 = enable interrupt. Read: 0 = interrupt disabled 1 = interrupt enabled.</td>
</tr>
</tbody>
</table>

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:

![ICER assignments](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>CLRENA</td>
<td>Interrupt clear-enable bits. Write:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = disable interrupt.</td>
</tr>
<tr>
<td></td>
<td>Read:</td>
<td>0 = interrupt disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = interrupt enabled</td>
</tr>
</tbody>
</table>

Table 4-5 ICER bit assignments

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:

![ISPR assignments](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>SETPEND</td>
<td>Interrupt set-pending bits. Write:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = changes interrupt state to pending.</td>
</tr>
<tr>
<td></td>
<td>Read:</td>
<td>0 = interrupt is not pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = interrupt is pending</td>
</tr>
</tbody>
</table>

Table 4-6 ISPR bit assignments

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-NVIC_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:

![CLRPEND bits]

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>CLRPEND</td>
<td>Interrupt clear-pending bits. Write: 0 = no effect 1 = removes pending state an interrupt. Read: 0 = interrupt is not pending 1 = interrupt is pending.</td>
</tr>
</tbody>
</table>

---

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 24 23 16 15 8 7 0
ACTIVE bits
```

Table 4-8 IABR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>ACTIVE</td>
<td>Interrupt active flags: 0 = interrupt not active 1 = interrupt active.</td>
</tr>
</tbody>
</table>

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 an 8-bit priority field for each interrupt and each register holds four priority fields. 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
I PR59 PRI_239 PRI_238 PRI_237 PRI_236
  :     :     :     :
I PRn PRI_4n+3 PRI_4n+2 PRI_4n+1 PRI_4n
  :     :     :     :
I PR0 PRI_3 PRI_2 PRI_1 PRI_0
```

Table 4-9 IPR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:24]</td>
<td>Priority, byte offset 3</td>
<td>Each implementation-defined priority field can hold a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. Register priority value fields are eight bits wide, and non-implemented low-order bits read as zero and ignore writes.</td>
</tr>
<tr>
<td>[23:16]</td>
<td>Priority, byte offset 2</td>
<td></td>
</tr>
<tr>
<td>[15:8]</td>
<td>Priority, byte offset 1</td>
<td></td>
</tr>
<tr>
<td>[7:0]</td>
<td>Priority, byte offset 0</td>
<td></td>
</tr>
</tbody>
</table>

See Accessing the Cortex-M4 NVIC registers using CMSIS on page 4-4 for more information about the access to the interrupt priority array, which 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 4-19.

**Note**

Only privileged software can enable unprivileged access to the STIR.

The bit assignments are:

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

**Table 4-10 STIR bit assignments**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Field</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:9]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[8:0]</td>
<td>INTID</td>
<td>Interrupt ID of the interrupt to trigger, in the range 0-239. For example, a value of 0x03 specifies interrupt IRQ3.</td>
</tr>
</tbody>
</table>

### 4.2.9 Level-sensitive and pulse interrupts

A Cortex-M4 device can support 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-M4 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, state of the interrupt changes to:
    - inactive, if the state was pending
    - active, if the state was active and pending.

4.2.10 NVIC usage 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.

A 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 setup for fault handlers, NMI and all enabled exception like interrupts. For more information see Vector Table Offset Register on page 4-16.

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:

```c
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

<table>
<thead>
<tr>
<th>CMSIS interrupt control function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void NVIC_SetPriorityGrouping(uint32_t priority_grouping)</td>
<td>Set the priority grouping</td>
</tr>
<tr>
<td>void NVIC_EnableIRQ(IRQn_t IRQn)</td>
<td>Enable IRQn</td>
</tr>
<tr>
<td>void NVIC_DisableIRQ(IRQn_t IRQn)</td>
<td>Disable IRQn</td>
</tr>
<tr>
<td>uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)</td>
<td>Return true (IRQ-Number) if IRQn is pending</td>
</tr>
<tr>
<td>void NVIC_SetPendingIRQ (IRQn_t IRQn)</td>
<td>Set IRQn pending</td>
</tr>
<tr>
<td>void NVIC_ClearPendingIRQ (IRQn_t IRQn)</td>
<td>Clear IRQn pending status</td>
</tr>
<tr>
<td>uint32_t NVIC_GetActive (IRQn_t IRQn)</td>
<td>Return the IRQ number of the active interrupt</td>
</tr>
<tr>
<td>void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)</td>
<td>Set priority for IRQn</td>
</tr>
<tr>
<td>uint32_t NVIC_GetPriority (IRQn_t IRQn)</td>
<td>Read priority of IRQn</td>
</tr>
<tr>
<td>void NVIC_SystemReset (void)</td>
<td>Reset the system</td>
</tr>
</tbody>
</table>

The input parameter IRQn is the IRQ number, see Table 2-16 on page 2-22. 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>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Required privilege</th>
<th>Reset value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E008</td>
<td>ACTLR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Auxiliary Control Register</td>
</tr>
<tr>
<td>0xE000E000</td>
<td>CPUID</td>
<td>RO</td>
<td>Privileged</td>
<td>0x410FC240</td>
<td>CPUI D Base Register on page 4-13</td>
</tr>
<tr>
<td>0xE000E004</td>
<td>ICSR</td>
<td>RWa</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Interrupt Control and State Register on page 4-13</td>
</tr>
<tr>
<td>0xE000E008</td>
<td>VTOR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Vector Table Offset Register on page 4-16</td>
</tr>
<tr>
<td>0xE000E00C</td>
<td>AIRCR</td>
<td>RWa</td>
<td>Privileged</td>
<td>0xFA050000</td>
<td>Application Interrupt and Reset Control Register on page 4-16</td>
</tr>
<tr>
<td>0xE000E10</td>
<td>SCR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>System Control Register on page 4-19</td>
</tr>
<tr>
<td>0xE000E14</td>
<td>CCR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000200</td>
<td>Configuration and Control Register on page 4-19</td>
</tr>
<tr>
<td>0xE000E18</td>
<td>SHPR1</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>System Handler Priority Register 1 on page 4-21</td>
</tr>
<tr>
<td>0xE000E1C</td>
<td>SHPR2</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>System Handler Priority Register 2 on page 4-22</td>
</tr>
<tr>
<td>0xE000E20</td>
<td>SHPR3</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>System Handler Priority Register 3 on page 4-22</td>
</tr>
<tr>
<td>0xE000E24</td>
<td>SHCRS</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>System Handler Control and State Register on page 4-23</td>
</tr>
<tr>
<td>0xE000E28</td>
<td>CFSR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Configurable Fault Status Register on page 4-24</td>
</tr>
<tr>
<td>0xE000E28</td>
<td>MMSRb</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00</td>
<td>MemManage Fault Status Register on page 4-25</td>
</tr>
<tr>
<td>0xE000E29</td>
<td>BFSRb</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00</td>
<td>BusFault Status Register on page 4-26</td>
</tr>
<tr>
<td>0xE000E2A</td>
<td>UFSRb</td>
<td>RW</td>
<td>Privileged</td>
<td>0x0000</td>
<td>UsageFault Status Register on page 4-28</td>
</tr>
<tr>
<td>0xE000E2C</td>
<td>HFSR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>HardFault Status Register on page 4-30</td>
</tr>
<tr>
<td>0xE000E34</td>
<td>MMAR</td>
<td>RW</td>
<td>Privileged</td>
<td>Unknown</td>
<td>MemManage Fault Address Register on page 4-30</td>
</tr>
<tr>
<td>0xE000E38</td>
<td>BFAR</td>
<td>RW</td>
<td>Privileged</td>
<td>Unknown</td>
<td>BusFault Address Register on page 4-31</td>
</tr>
<tr>
<td>0xE000E3C</td>
<td>AFSR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Auxiliary Fault Status Register on page 4-31</td>
</tr>
</tbody>
</table>

a. See the register description for more information.
b. A subregister of the CFSR.

4.3.1 Auxiliary Control Register

The ACTLR provides disable bits for the following processor functions:

• IT folding
• write buffer use for accesses to the default memory map
• interruption of multi-cycle instructions.

By default this register is set to provide optimum performance from the Cortex-M4 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:

![ACTLR bit assignments diagram]

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:10]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[9]</td>
<td>DISOOFP&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Disables floating point instructions completing out of order with respect to integer instructions.</td>
</tr>
<tr>
<td>[8]</td>
<td>DISFPCA&lt;sup&gt;a&lt;/sup&gt;</td>
<td>Disables automatic update of CONTROL.FPCA.</td>
</tr>
<tr>
<td>[7:3]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[2]</td>
<td>DISFOLD</td>
<td>When set to 1, disables IT folding. see About IT folding for more information.</td>
</tr>
<tr>
<td>[1]</td>
<td>DISDEFWBUF</td>
<td>When set to 1, disables write buffer use during default memory map accesses. This causes all BusFaults to be precise BusFaults but decreases performance because any store to memory must complete before the processor can execute the next instruction.</td>
</tr>
</tbody>
</table>

**Note**

This bit only affects write buffers implemented in the Cortex-M4 processor.

| [0] | DISMCYCINT        | When set to 1, disables interruption of load multiple and store multiple instructions. This increases the interrupt latency of the processor because any LDM or STM must complete before the processor can stack the current state and enter the interrupt handler. |

---

<sup>a</sup> Only implemented in a Cortex-M4F device

### About IT folding

In some situations, the processor can start executing the first instruction in an IT block while it is still executing the IT instruction. This behavior is called IT folding, and improves performance, However, IT folding can cause jitter in looping. If a task must avoid jitter, set the DISFOLD bit to 1 before executing the task, to disable IT folding.
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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:24</td>
<td>Implementer</td>
<td>Implementer code: 0x41 = ARM</td>
</tr>
<tr>
<td>23:20</td>
<td>Variant</td>
<td>Variant number, the r value in the rpmn product revision identifier: 0x0 = Revision 0</td>
</tr>
<tr>
<td>19:16</td>
<td>Constant</td>
<td>Reads as 0xF</td>
</tr>
<tr>
<td>15:4</td>
<td>PartNo</td>
<td>Part number of the processor: 0xC24 = Cortex-M4</td>
</tr>
<tr>
<td>3:0</td>
<td>Revision</td>
<td>Revision number, the p value in the rpmn product revision identifier: 0x0 = Patch 0</td>
</tr>
</tbody>
</table>

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, for the ICSR attributes. The bit assignments are:

![Bit assignments diagram](image-url)

**Table 4-15 ICSR bit assignments**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
</table>
          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 enter 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. |
          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. |
          Write:  
          0 = no effect  
          1 = changes SysTick exception state to pending.  
          Read:  
          0 = SysTick exception is not pending  
          1 = SysTick exception is pending. |
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.

Table 4-15 ICSR bit assignments (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[25]</td>
<td>PENDSTCLR</td>
<td>WO</td>
<td>SysTick exception clear-pending bit. Write:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = no effect</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = removes the pending state from the SysTick exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit is WO. On a register read its value is unknown.</td>
</tr>
<tr>
<td>[23]</td>
<td>Reserved for</td>
<td>RO</td>
<td>This bit is reserved for Debug use and reads-as-zero when the processor is not in Debug.</td>
</tr>
<tr>
<td></td>
<td>Debug use</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[22]</td>
<td>ISRPENDING</td>
<td>RO</td>
<td>Interrupt pending flag, excluding NMI and Faults:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = interrupt not pending</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = interrupt pending.</td>
</tr>
<tr>
<td>[17:12]</td>
<td>VECTPENDING</td>
<td>RO</td>
<td>Indicates the exception number of the highest priority pending enabled exception:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = no pending exceptions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Nonzero = the exception number of the highest priority pending enabled exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The value indicated by this field includes the effect of the BASEPRI and FAULTMASK registers, but not any effect of the PRIMASK register.</td>
</tr>
<tr>
<td>[11]</td>
<td>RETTOBASE</td>
<td>RO</td>
<td>Indicates whether there are preempted active exceptions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = there are preempted active exceptions to execute</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = there are no active exceptions, or the currently-executing exception is the only active exception.</td>
</tr>
<tr>
<td>[8:0]</td>
<td>VECTACTIVE&lt;sup&gt;a&lt;/sup&gt;</td>
<td>RO</td>
<td>Contains the active exception number:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Thread mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Nonzero = The exception number&lt;sup&gt;a&lt;/sup&gt; of the currently active exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Note</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

<sup>a</sup> This is the same value as IPSR bits[8:0], see Interrupt Program Status Register on page 2-6.
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 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| TBLOFF | Reserved |
```

Table 4-16 VTOR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>

**Note**

Bit[29] determines whether the vector table is in the code or SRAM memory region:

- 0 = code
- 1 = SRAM.

In implementations bit[29] is sometimes called the TBLBASE bit.

| [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. See your vendor documentation for the alignment details of your device.

**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 4-17 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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:16]</td>
<td>Write: VECTKEYSTAT</td>
<td>RW</td>
<td>Register key:</td>
</tr>
<tr>
<td></td>
<td>Read: VECTKEY</td>
<td></td>
<td>Reads as 0xFA05</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>On writes, write 0x5FA to VECTKEY, otherwise the write is ignored.</td>
</tr>
<tr>
<td>[15]</td>
<td>ENDIANNESS</td>
<td>RO</td>
<td>Data endianness bit is implementation defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = Little-endian</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = Big-endian</td>
</tr>
<tr>
<td>[10:8]</td>
<td>PRIGROUP</td>
<td>R/W</td>
<td>Interrupt priority grouping field is implementation defined. This field</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>determines the split of group priority from subpriority, see</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7:3]</td>
<td>-</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[2]</td>
<td>SYSRESETREQ</td>
<td>WO</td>
<td>System reset request bit is implementation defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 = no system reset request</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 = asserts a signal to the outer system that requests a reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This is intended to force a large system reset of all major components</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>except for debug.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bit reads as 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See your vendor documentation for more information about the use of this</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>signal in your implementation.</td>
</tr>
<tr>
<td>[1]</td>
<td>VECTCLRACTIVE</td>
<td>WO</td>
<td>Reserved for Debug use. This bit reads as 0. When writing to the register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>you must write 0 to this bit, otherwise behavior is Unpredictable.</td>
</tr>
<tr>
<td>[0]</td>
<td>VECTRESET</td>
<td>WO</td>
<td>Reserved for Debug use. This bit reads as 0. When writing to the register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>you must write 0 to this bit, otherwise behavior is Unpredictable.</td>
</tr>
</tbody>
</table>
Binary point

The PRIGROUP field indicates the position of the binary point that splits the PRIₙ 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 fewer than 8-bits of interrupt priority treat the least significant bits as zero.

<table>
<thead>
<tr>
<th>PRIGROUP</th>
<th>Binary point¹</th>
<th>Group priority bits</th>
<th>Subpriority bits</th>
<th>Group priorities</th>
<th>Subpriorities</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>bxxxxxx.y</td>
<td>[7:1]</td>
<td>[0]</td>
<td>128</td>
<td>2</td>
</tr>
<tr>
<td>0b001</td>
<td>bxxxxx.yy</td>
<td>[7:2]</td>
<td>[1:0]</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>0b010</td>
<td>bxxxx.yyy</td>
<td>[7:3]</td>
<td>[2:0]</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>0b011</td>
<td>bxxx.yyyyy</td>
<td>[7:4]</td>
<td>[3:0]</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>0b100</td>
<td>bxx.yyyyy</td>
<td>[7:5]</td>
<td>[4:0]</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>0b101</td>
<td>bx.yyyyyy</td>
<td>[7:6]</td>
<td>[5:0]</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>0b110</td>
<td>bx.yyyyyyy</td>
<td>[7]</td>
<td>[6:0]</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>0b111</td>
<td>b.yyyyyyyyy</td>
<td>None</td>
<td>[7:0]</td>
<td>1</td>
<td>256</td>
</tr>
</tbody>
</table>

¹ PRIₙ[7:0] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

**Note**

Determining preemption of an exception uses only the group priority field, see *Interrupt priority grouping* on page 2-25.
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:

![SCR Bit Assignments Diagram]

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:5]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[4]</td>
<td>SEVONPEND</td>
<td>Send Event on Pending bit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = only enabled interrupts or events can wakeup the processor, disabled interrupts are excluded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = enabled events and all interrupts, including disabled interrupts, can wakeup the processor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The processor also wakes up on execution of an SEV instruction or an external event.</td>
</tr>
<tr>
<td>[2]</td>
<td>SLEEPDEEP</td>
<td>Controls whether the processor uses sleep or deep sleep as its low power mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = sleep</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = deep sleep.</td>
</tr>
<tr>
<td>[1]</td>
<td>SLEEPONEXIT</td>
<td>Indicates sleep-on-exit when returning from Handler mode to Thread mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = do not sleep when returning to Thread mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = enter sleep, or deep sleep, on return from an ISR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application.</td>
</tr>
<tr>
<td>[0]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

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.

See the register summary in Table 4-12 on page 4-11 for the CCR attributes.
The bit assignments are:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:10]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[9]</td>
<td>STKALIGN</td>
<td>Indicates stack alignment on exception entry:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = 4-byte aligned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = 8-byte aligned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[8]</td>
<td>BFHFNIGN</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = data bus faults caused by load and store instructions cause a lock-up</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = handlers running at priority -1 and -2 ignore data bus faults caused by load and store instructions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[7:5]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[4]</td>
<td>DIV_0_TRP</td>
<td>Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = do not trap divide by 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = trap divide by 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this bit is set to 0, a divide by zero returns a quotient of 0.</td>
</tr>
<tr>
<td>[3]</td>
<td>UNALIGN_TRP</td>
<td>Enables unaligned access traps:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = do not trap unaligned halfword and word accesses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = trap unaligned halfword and word accesses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this bit is set to 1, an unaligned access generates a UsageFault.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of whether UNALIGN_TRP is set to 1.</td>
</tr>
</tbody>
</table>
4.3.8 System Handler Priority Registers

The SHPR1-SHPR3 registers set the priority level, 0 to 255, of the exception handlers that have configurable priority.

SHPR1-SHPR3 are byte accessible. See the register summary in Table 4-12 on page 4-11 for their attributes.

To access to the system exception priority level using CMSIS, use the following CMSIS functions:

- `uint32_t NVIC_GetPriority(IRQn_Type IRQn)`
- `void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

The input parameter IRQn is the IRQ number, see Table 2-16 on page 2-22 for more information.

### System Handler Priority Register 1

The bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[23:16]</td>
<td>PRI_6</td>
<td>Priority of system handler 6, UsageFault</td>
</tr>
<tr>
<td>[15:8]</td>
<td>PRI_5</td>
<td>Priority of system handler 5, BusFault</td>
</tr>
<tr>
<td>[7:0]</td>
<td>PRI_4</td>
<td>Priority of system handler 4, MemManage</td>
</tr>
</tbody>
</table>

![Table 4-20 CCR bit assignments (continued)](image-url)
System Handler Priority Register 2

The bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRI_11</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-22 SHPR2 register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:24]</td>
<td>PRI_11</td>
<td>Priority of system handler 11, SVCall</td>
</tr>
<tr>
<td>[23:0]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

System Handler Priority Register 3

The bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRI_15</td>
<td>PRI_14</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-23 SHPR3 register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:24]</td>
<td>PRI_15</td>
<td>Priority of system handler 15, SysTick exception</td>
</tr>
<tr>
<td>[23:16]</td>
<td>PRI_14</td>
<td>Priority of system handler 14, PendSV</td>
</tr>
<tr>
<td>[15:0]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
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 Table 4-12 on page 4-11 for the SHCSR attributes. The bit assignments are:

![Table 4-24 SHCSR bit assignments](image)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:19]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[18]</td>
<td>USGFAULTENA</td>
<td>UsageFault enable bit, set to 1 to enable(^a)</td>
</tr>
<tr>
<td>[17]</td>
<td>BUSFAULTENA</td>
<td>BusFault enable bit, set to 1 to enable(^a)</td>
</tr>
<tr>
<td>[16]</td>
<td>MEMFAULTENA</td>
<td>MemManage enable bit, set to 1 to enable(^a)</td>
</tr>
<tr>
<td>[15]</td>
<td>SVCALLPENDED</td>
<td>SVCall pending bit, reads as 1 if exception is pending(^b)</td>
</tr>
<tr>
<td>[14]</td>
<td>BUSFAULTPENDED</td>
<td>BusFault exception pending bit, reads as 1 if exception is pending(^b)</td>
</tr>
<tr>
<td>[13]</td>
<td>MEMFAULTPENDED</td>
<td>MemManage exception pending bit, reads as 1 if exception is pending(^b)</td>
</tr>
<tr>
<td>[12]</td>
<td>USGFAULTPENDED</td>
<td>UsageFault exception pending bit, reads as 1 if exception is pending(^b)</td>
</tr>
<tr>
<td>[11]</td>
<td>SYSTICKACT</td>
<td>SysTick exception active bit, reads as 1 if exception is active(^c)</td>
</tr>
<tr>
<td>[10]</td>
<td>PENDSVACT</td>
<td>PendSV exception active bit, reads as 1 if exception is active</td>
</tr>
<tr>
<td>[8]</td>
<td>MONITORACT</td>
<td>Debug monitor active bit, reads as 1 if Debug monitor is active</td>
</tr>
<tr>
<td>[7]</td>
<td>SVCALLACT</td>
<td>SVCall active bit, reads as 1 if SVC call is active</td>
</tr>
<tr>
<td>[3]</td>
<td>USGFAULTACT</td>
<td>UsageFault exception active bit, reads as 1 if exception is active</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>BUSFAULTACT</td>
<td>BusFault exception active bit, reads as 1 if exception is active</td>
</tr>
<tr>
<td>[0]</td>
<td>MEMFAULTACT</td>
<td>MemManage exception active bit, reads as 1 if exception is active</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Usage Fault Status Register</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Bus Fault Status Register</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Memory Management Fault Status Register</td>
<td></td>
</tr>
</tbody>
</table>

The following subsections describe the subregisters that make up the CFSR:

- MemManage Fault Status Register on page 4-25
- BusFault Status Register on page 4-26
- UsageFault Status Register on page 4-28.

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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>MMARVALID</td>
<td>MemManage Fault Address Register (MMFAR) valid flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = value in MMAR is not a valid fault address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = MMAR holds a valid fault address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[5]</td>
<td>MLSPERRa</td>
<td>0 = no MemManage fault occurred during floating-point lazy state preservation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = a MemManage fault occurred during floating-point lazy state preservation.</td>
</tr>
<tr>
<td>[4]</td>
<td>MSTKERR</td>
<td>MemManage fault on stacking for exception entry:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no stacking fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = stacking for an exception entry has caused one or more access violations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[3]</td>
<td>MUNSTKERR</td>
<td>MemManage fault on unstacking for a return from exception:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no unstacking fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = unstack for an exception return has caused one or more access violations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[1]</td>
<td>DACCVIOL</td>
<td>Data access violation flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no data access violation fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor attempted a load or store at a location that does not permit the operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[0]</td>
<td>IACCVIOL</td>
<td>Instruction access violation flag:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no instruction access violation fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor attempted an instruction fetch from a location that does not permit execution.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This fault occurs on any access to an XN region, even when the MPU is disabled or not present.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

a. Only present in a Cortex-M4F device.
BusFault Status Register

The flags in the BFSR indicate the cause of a bus access fault. The bit assignments are:

![BusFault Status Register Diagram]

Table 4-26 BFSR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>BFARVALID</td>
<td><strong>BusFault Address Register (BFAR) valid flag:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = value in BFAR is not a valid fault address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = BFAR holds a valid fault address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The processor sets this bit to 1 after a BusFault where the address is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>known. Other faults can set this bit to 0, such as a MemManage fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>occurring later.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If a BusFault occurs and is escalated to a hard fault because of priority,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the hard fault handler must set this bit to 0. This prevents problems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if returning to a stacked active BusFault handler whose BFAR value has</td>
</tr>
<tr>
<td></td>
<td></td>
<td>been overwritten.</td>
</tr>
<tr>
<td>[5]</td>
<td>LSPERRa</td>
<td>0 = no bus fault occurred during floating-point lazy state preservation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = a bus fault occurred during floating-point lazy state preservation.</td>
</tr>
<tr>
<td>[4]</td>
<td>STKERR</td>
<td>BusFault on stacking for exception entry:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no stacking fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = stacking for an exception entry has caused one or more BusFaults.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the processor sets this bit to 1, the SP is still adjusted but the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values in the context area on the stack might be incorrect. The processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>does not write a fault address to the BFAR.</td>
</tr>
<tr>
<td>[3]</td>
<td>UNSTKERR</td>
<td>BusFault on unstacking for a return from exception:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no unstacking fault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = unstack for an exception return has caused one or more BusFaults.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This fault is chained to the handler. This means that when the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sets this bit to 1, the original return stack is still present. The</td>
</tr>
<tr>
<td></td>
<td></td>
<td>processor does not adjust the SP from the failing return, does not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>performed a new save, and does not write a fault address to the BFAR.</td>
</tr>
</tbody>
</table>
### 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.

### 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 is 1, it writes the faulting address to the BFAR.

### 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 is 1, it does not write a fault address to the BFAR.

---

**Table 4-26 BFSR bit assignments (continued)**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>IMPRECISERR</td>
<td>Imprecise data bus error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no imprecise data bus error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the processor sets this bit to 1, it does not write a fault address to the BFAR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[1]</td>
<td>PRECISERR</td>
<td>Precise data bus error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no precise data bus error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = a data bus error has occurred, and the PC value stacked for the exception return points to the instruction that caused the fault.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the processor sets this bit is 1, it writes the faulting address to the BFAR.</td>
</tr>
<tr>
<td>[0]</td>
<td>IBUSERR</td>
<td>Instruction bus error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no instruction bus error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = instruction bus error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the processor sets this bit is 1, it does not write a fault address to the BFAR.</td>
</tr>
</tbody>
</table>

a. Only present in a Cortex-M4F device.
UsageFault Status Register

The UFSR indicates the cause of a UsageFault. The bit assignments are:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[15:10]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[9]</td>
<td>DIVBYZERO</td>
<td>Divide by zero UsageFault:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no divide by zero fault, or divide by zero trapping not enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor has executed an SDIV or UDIV instruction with a divisor of 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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 4-19.</td>
</tr>
<tr>
<td>[8]</td>
<td>UNALIGNED</td>
<td>Unaligned access UsageFault:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no unaligned access fault, or unaligned access trapping not enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor has made an unaligned memory access.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Enable trapping of unaligned accesses by setting the UNALIGN_TRP bit in the CCR to 1, see Configuration and Control Register on page 4-19.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of the setting of UNALIGN_TRP.</td>
</tr>
<tr>
<td>[3]</td>
<td>NOCP</td>
<td>No coprocessor UsageFault. The processor does not support coprocessor instructions:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = no UsageFault caused by attempting to access a coprocessor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor has attempted to access a coprocessor.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>INVPC</td>
<td>Invalid PC load UsageFault, caused by an invalid PC load by EXC_RETURN: 0 = no invalid PC load UsageFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[1]</td>
<td>INVSTATE</td>
<td>Invalid state UsageFault: 0 = no invalid state UsageFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor has attempted to execute an instruction that makes illegal use of the EPSR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This bit is not set to 1 if an undefined instruction uses the EPSR.</td>
</tr>
<tr>
<td>[0]</td>
<td>UNDEFINSTR</td>
<td>Undefined instruction UsageFault: 0 = no undefined instruction UsageFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = the processor has attempted to execute an undefined instruction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>
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:

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

FORCED
DEBUGEVT

Reserved

Table 4-28 HFSR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>DEBUGEVT</td>
<td>Reserved for Debug use. When writing to the register you must write 0 to this bit, otherwise behavior is Unpredictable.</td>
</tr>
</tbody>
</table>
| [30] | FORCED  | Indicates a forced hard fault, generated by escalation of a fault with configurable priority that cannot be handles, 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 4-29 MMFAR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>ADDRESS</td>
<td>When the MMARVALID bit of the MMFSR is set to 1, this field holds the address of the location that generated the MemManage fault</td>
</tr>
</tbody>
</table>
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 4-25.

### 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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>ADDRESS</td>
<td>When the BFARVALID bit of the BFSR is set to 1, this field holds the address of the location that generated the BusFault</td>
</tr>
</tbody>
</table>

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 4-26.

### 4.3.14 Auxiliary Fault Status Register

The AFSR contains additional system fault information. 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>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:0]</td>
<td>IMPDEF</td>
<td>Implementation defined. The bits map to the AUXFAULT input signals.</td>
</tr>
</tbody>
</table>

Each AFSR bit maps directly to an AUXFAULT input of the processor, and a single-cycle HIGH signal on the input sets the corresponding AFSR bit to one. It remains set to 1 until you write 1 to the bit to clear it to zero. See your vendor documentation for more information.

When an AFSR bit is latched as one, an exception does not occur. Use an interrupt if an exception is required.

### 4.3.15 System control block usage 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.
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:

Table 4-32 System timer registers summary

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Required privilege</th>
<th>Reset value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000E010</td>
<td>SYST_CSR</td>
<td>RW</td>
<td>Privileged</td>
<td>a</td>
<td>SysTick Control and Status Register</td>
</tr>
<tr>
<td>0xE000E014</td>
<td>SYST_RVR</td>
<td>RW</td>
<td>Privileged</td>
<td>Unknown</td>
<td>SysTick Reload Value Register on page 4-34</td>
</tr>
<tr>
<td>0xE000E018</td>
<td>SYST_CVR</td>
<td>RW</td>
<td>Privileged</td>
<td>Unknown</td>
<td>SysTick Current Value Register on page 4-35</td>
</tr>
<tr>
<td>0xE000E01C</td>
<td>SYST_CALIB</td>
<td>RO</td>
<td>Privileged</td>
<td>- a</td>
<td>SysTick Calibration Value Register on page 4-35</td>
</tr>
</tbody>
</table>

a. See the register description for more information.

4.4.1 SysTick Control and Status Register

The SysTick SYST_CSR register enables the SysTick features. The register resets to 0x00000000, or to 0x00000004 if your device does not implement a reference clock. See the register summary in Table 4-32 for its attributes. The bit assignments are:

Table 4-33 SysTick SYST_CSR register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:17]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[16]</td>
<td>COUNTFLAG</td>
<td>Returns 1 if timer counted to 0 since last time this was read.</td>
</tr>
<tr>
<td>[15:3]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
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 Table 4-32 on page 4-33 for its attributes. The bit assignments are:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:24]</td>
<td>RELOAD</td>
<td>Value to load into the SYST_CVR register when the counter is enabled and when it reaches 0, see Calculating the RELOAD value.</td>
</tr>
<tr>
<td>[23:0]</td>
<td>RELOAD</td>
<td>Value to load into the SYST_CVR register when the counter is enabled and when it reaches 0, see Calculating the RELOAD value.</td>
</tr>
</tbody>
</table>

#### Calculating the RELOAD value

The RELOAD value can be any value in the range 0x00000001-0xFFFFFFFF. 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 4-32 on page 4-33 for its attributes. The bit assignments are:

```
31 | 24 | 23 |          |          | 0
   | Reserved |          | CURRENT  |
```

Table 4-35 SYST_CVR register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:24]</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>
| [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

The SYST_CALIB register indicates the SysTick calibration properties. See the register summary in Table 4-32 on page 4-33 for its attributes. The reset value of this register is implementation-defined. See the documentation supplied by your device vendor for more information about the meaning of the SYST_CALIB field values. The bit assignments are:

```
31 30 29 | 24 | 23 |          |          | 0
   | Reserved |          | TENMS    |
       | SKEW     | NOREF    |          |
```

Table 4-36 SYST_CALIB register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| [31]   | NOREF    | Indicates whether the device provides a reference clock to the processor:  
          | 0 = reference clock provided  
          | 1 = no reference clock provided.  
          | If your device does not provide a reference clock, the SYST_CSR.CLKSOURCE bit reads-as-one and ignores writes. |
| [30]   | SKEW     | Indicates whether the TENMS value is exact:  
          | 0 = TENMS value is exact  
          | 1 = TENMS value is inexact, or not given.  
          | An inexact TENMS value can affect the suitability of SysTick as a software real time clock. |
| [29:24]| Reserved |                                                |
| [23:0]| TENMS    | Reload value for 10ms (100Hz) timing, subject to system clock skew errors. If the value reads as zero, the calibration value is not known. |

If calibration information is not known, calculate the calibration value required from the frequency of the processor clock or external clock.
4.4.5 SysTick usage hints and tips

Some implementations stop all the processor clock signals during deep sleep mode. If this happens, the SysTick counter stops.

Ensure software uses aligned word accesses to access the SysTick registers.

The SysTick counter reload and current value are not initialized by hardware. This means the correct initialization sequence for the SysTick counter is:
1. Program reload value.
2. Clear current value.
3. Program Control and Status register.
4.5 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-M4 MPU defines:

- eight separate memory regions, 0-7
- a 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-M4 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 4-37 shows the possible MPU region attributes. These include Shareability and cache behavior attributes are not relevant to most microcontroller implementations. See MPU configuration for a microcontroller on page 4-47 and your vendor documentation for programming guidelines if implemented.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Shareability</th>
<th>Other attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly-ordered</td>
<td>-</td>
<td>-</td>
<td>All accesses to Strongly-ordered memory occur in program order. All Strongly-ordered regions are assumed to be shared.</td>
</tr>
<tr>
<td>Device</td>
<td>Shared</td>
<td>-</td>
<td>Memory-mapped peripherals that several processors share.</td>
</tr>
<tr>
<td>Non-shared</td>
<td>-</td>
<td>-</td>
<td>Memory-mapped peripherals that only a single processor uses.</td>
</tr>
<tr>
<td>Normal</td>
<td>Shared</td>
<td>Non-cacheable Write-through Cacheable Write-back Cacheable</td>
<td>Normal memory that is shared between several processors.</td>
</tr>
<tr>
<td>Non-shared</td>
<td>Non-cacheable Write-through Cacheable Write-back Cacheable</td>
<td>Normal memory that only a single processor uses.</td>
<td></td>
</tr>
</tbody>
</table>
Use the MPU registers to define the MPU regions and their attributes. The MPU registers are:

Table 4-38 MPU registers summary

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Required privilege</th>
<th>Reset value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000ED90</td>
<td>MPU_TYPE</td>
<td>RO</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>MPU Type Register</td>
</tr>
<tr>
<td>0xE000ED94</td>
<td>MPU_CTRL</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>MPU Control Register on page 4-39</td>
</tr>
<tr>
<td>0xE000ED98</td>
<td>MPU_RNR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>MPU Region Number Register on page 4-40</td>
</tr>
<tr>
<td>0xE000ED9C</td>
<td>MPU_RBAR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>MPU Region Base Address Register on page 4-40</td>
</tr>
<tr>
<td>0xE000EDA0</td>
<td>MPU_RASR</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>MPU Region Attribute and Size Register on page 4-41</td>
</tr>
<tr>
<td>0xE000EDA4</td>
<td>MPU_RBAR_A1</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RBAR, see MPU Region Base Address Register on page 4-40</td>
</tr>
<tr>
<td>0xE000EDA8</td>
<td>MPU_RASR_A1</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RASR, see MPU Region Attribute and Size Register on page 4-41</td>
</tr>
<tr>
<td>0xE000EDAC</td>
<td>MPU_RBAR_A2</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RBAR, see MPU Region Base Address Register on page 4-40</td>
</tr>
<tr>
<td>0xE000EDB0</td>
<td>MPU_RASR_A2</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RASR, see MPU Region Attribute and Size Register on page 4-41</td>
</tr>
<tr>
<td>0xE000EDB4</td>
<td>MPU_RBAR_A3</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RBAR, see MPU Region Base Address Register on page 4-40</td>
</tr>
<tr>
<td>0xE000EDB8</td>
<td>MPU_RASR_A3</td>
<td>RW</td>
<td>Privileged</td>
<td>0x00000000</td>
<td>Alias of RASR, see MPU Region Attribute and Size Register on page 4-41</td>
</tr>
</tbody>
</table>

4.5.1 MPU Type Register

The MPU_TYPE register indicates whether the MPU is present, and if so, how many regions it supports. See the register summary in Table 4-38 for its attributes. The bit assignments are:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| [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:  
  0x08 = Eight MPU regions.                                                                                           |
| [7:1] | -    | Reserved.                                                                                                                             |
| [0]   | SEPARATE | Indicates support for unified or separate instruction and date memory maps:  
  0 = unified.                                                                                                    |
4.5.2 MPU Control Register

The MPU_CTRL register:

- enables the MPU
- enables the default 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 4-38 on page 4-38 for the MPU_CTRL attributes. The bit assignments are:

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

Table 4-40 MPU_CTRL register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| [31:3]| Reserved  | 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.

### 4.5.3 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 Table 4-38 on page 4-38 for its attributes.

The bit assignments are:

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Region</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:8]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[7:0]</td>
<td>REGION</td>
<td>Indicates the MPU region referenced by the MPU_RBAR and MPU_RASR registers. The MPU supports 8 memory regions, so the permitted values of this field are 0-7.</td>
</tr>
</tbody>
</table>

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.

### 4.5.4 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 Table 4-38 on page 4-38 for its attributes.
Write MPU_RBAR with the VALID bit set to 1 to change the current region number and update the MPU_RNR. The bit assignments are:

```
31 |   |   | N | N-1 | 5 | 4 | 3 | 0
   | ADDR|   |   |     |   |   |   | VALID
   |     | Reserved | REGION
```

If the region size is 32B, the ADDR field is bits [31:5] and there is no Reserved field.

### Table 4-42 MPU_RBAR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:N] ADDR</td>
<td>Region base address field. The value of N depends on the region size. For more information see The ADDR field.</td>
<td></td>
</tr>
<tr>
<td>[N-1]:5</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[4] VALID</td>
<td>MPU Region Number valid bit: Write: 0 = MPU_RNR not changed, and the processor: • updates the base address for the region specified in the MPU_RNR • ignores the value of the REGION field 1 = the processor: • 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.</td>
<td></td>
</tr>
<tr>
<td>[3:0] REGION</td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>

#### 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:

\[ N = \log_2(\text{Region size in bytes}) \]

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.

#### 4.5.5 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 4-38 on page 4-38 for its attributes.

MPU_RASR is accessible using word or halfword accesses:

- the most significant word holds the region attributes
- the least significant halfword holds the region size and the region and subregion enable bits.
The bit assignments are:

```
<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AP</td>
<td>TEX</td>
<td>S</td>
<td>C</td>
<td>B</td>
<td>SRD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>XN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:29]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[28]</td>
<td>XN</td>
<td>Instruction access disable bit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = instruction fetches enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = instruction fetches disabled</td>
</tr>
<tr>
<td>[27]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
<tr>
<td>[26:24]</td>
<td>AP</td>
<td>Access permission field, see Table 4-47 on page 4-44.</td>
</tr>
<tr>
<td>[21:19,17,16]</td>
<td>TEX, C, B</td>
<td>Memory access attributes, see Table 4-45 on page 4-43.</td>
</tr>
<tr>
<td>[18]</td>
<td>S</td>
<td>Shareable bit, see Table 4-45 on page 4-43.</td>
</tr>
<tr>
<td>[15:8]</td>
<td>SRD</td>
<td>Subregion disable bits. For each bit in this field:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = corresponding sub-region is enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = corresponding sub-region is disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Subregions on page 4-46 for more information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>[5:1]</td>
<td>SIZE</td>
<td>Specifies the size of the MPU protection region. The minimum permitted value is 3 (0b000010). See SIZE field values for more information.</td>
</tr>
<tr>
<td>[0]</td>
<td>ENABLE</td>
<td>Region enable bit.</td>
</tr>
</tbody>
</table>

For information about access permission, see MPU access permission attributes on page 4-43.

**SIZE field values**

The SIZE field defines the size of the MPU memory region specified by the RNR. as follows:

\[(\text{Region size in bytes}) = 2^{(\text{SIZE}+1)}\]
The smallest permitted region size is 32B, corresponding to a SIZE value of 4. Table 4-44 gives example SIZE values, with the corresponding region size and value of N in the MPU_RBAR.

### Table 4-44 Example SIZE field values

<table>
<thead>
<tr>
<th>SIZE value</th>
<th>Region size</th>
<th>Value of N</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0100 (4)</td>
<td>32B</td>
<td>5</td>
<td>Minimum permitted size</td>
</tr>
<tr>
<td>0b1001 (9)</td>
<td>1KB</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>0b10011 (19)</td>
<td>1MB</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>0b11101 (29)</td>
<td>1GB</td>
<td>30</td>
<td>-</td>
</tr>
<tr>
<td>0b11111 (31)</td>
<td>4GB</td>
<td>32</td>
<td>Maximum possible size</td>
</tr>
</tbody>
</table>

Note: In the MPU_RBAR, see MPU Region Base Address Register on page 4-40.

### 4.5.6 MPU access permission attributes

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 4-45 shows encodings for the TEX, C, B, and S access permission bits.

### Table 4-45 TEX, C, B, and S encoding

<table>
<thead>
<tr>
<th>TEX</th>
<th>C</th>
<th>B</th>
<th>S</th>
<th>Memory type</th>
<th>Shareability</th>
<th>Other attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b000</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>Strongly-ordered</td>
<td>Shareable</td>
<td>-</td>
</tr>
<tr>
<td>0b001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Normal</td>
<td>Not shareable</td>
<td>Outer and inner write-through. No write allocate.</td>
</tr>
<tr>
<td>0b010</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>Device</td>
<td>Not shareable</td>
<td>Outer and inner noncacheable.</td>
</tr>
<tr>
<td>0b101</td>
<td>A</td>
<td>A</td>
<td>0</td>
<td>Normal</td>
<td>Not shareable</td>
<td>Outer and inner write-back. Write and read allocate.</td>
</tr>
</tbody>
</table>

Note: The MPU ignores the value of this bit.
Table 4-46 shows the cache policy for memory attribute encodings with a TEX value is in the range 4-7.

<table>
<thead>
<tr>
<th>Encoding, AA or BB</th>
<th>Corresponding cache policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Non-cacheable</td>
</tr>
<tr>
<td>01</td>
<td>Write back, write and read allocate</td>
</tr>
<tr>
<td>10</td>
<td>Write through, no write allocate</td>
</tr>
<tr>
<td>11</td>
<td>Write back, no write allocate</td>
</tr>
</tbody>
</table>

Table 4-47 shows the AP encodings that define the access permissions for privileged and unprivileged software.

<table>
<thead>
<tr>
<th>AP[2:0]</th>
<th>Privileged permissions</th>
<th>Unprivileged permissions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>No access</td>
<td>No access</td>
<td>All accesses generate a permission fault</td>
</tr>
<tr>
<td>001</td>
<td>RW</td>
<td>No access</td>
<td>Access from privileged software only</td>
</tr>
<tr>
<td>010</td>
<td>RW</td>
<td>RO</td>
<td>Writes by unprivileged software generate a permission fault</td>
</tr>
<tr>
<td>011</td>
<td>RW</td>
<td>RW</td>
<td>Full access</td>
</tr>
<tr>
<td>100</td>
<td>Unpredictable</td>
<td>Unpredictable</td>
<td>Reserved</td>
</tr>
<tr>
<td>101</td>
<td>RO</td>
<td>No access</td>
<td>Reads by privileged software only</td>
</tr>
<tr>
<td>110</td>
<td>RO</td>
<td>RO</td>
<td>Read only, by privileged or unprivileged software</td>
</tr>
<tr>
<td>111</td>
<td>RO</td>
<td>RO</td>
<td>Read only, by privileged or unprivileged software</td>
</tr>
</tbody>
</table>

4.5.7 MPU mismatch

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 4-25 for more information.

4.5.8 Updating an MPU region

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       
      ; 0x0000ED98, MPU region number register
```
STR R1, [R0, #0x0] ; Region Number
STR R4, [R0, #0x4] ; Region Base Address
STRH R2, [R0, #0x8] ; Region Size and Enable
STRH R3, [R0, #0xA] ; Region Attribute

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
BIC R2, R2, #1 ; Disable
STRH R2, [R0, #0x8] ; Region Size and Enable
STR R4, [R0, #0x4] ; Region Base Address
STRH R3, [R0, #0xA] ; Region Attribute
ORR R2, #1 ; Enable
STRH R2, [R0, #0x8] ; Region Size and Enable

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.

However, memory barrier instructions are not required if the MPU setup process starts by entering an exception handler, or is followed by an exception return, because the exception entry and exception return mechanism cause memory barrier behavior.

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 using a return from exception, or by taking an exception, then you do not require an ISB.

**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 4-40. 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 4-41. 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 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 0x0, 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.

```
Region 2, with subregions
Offset from base address
512KB
448KB
384KB
320KB
256KB
192KB
128KB
64KB
0
```

<table>
<thead>
<tr>
<th></th>
<th>Region 1</th>
<th>Disabled subregion</th>
<th>Disabled subregion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base address of both regions</td>
<td>0</td>
<td>64KB</td>
<td>128KB</td>
</tr>
</tbody>
</table>

**4.5.9 MPU usage hints and tips**

To avoid unexpected behavior, disable the interrupts before updating the attributes of a region that the interrupt handlers might access.

Ensure software uses aligned accesses of the correct size to access MPU registers:
- except for the MPU_RASR, it must use aligned word accesses
- for the MPU_RASR it can use byte or aligned halfword or word accesses.

The processor does not support unaligned accesses to MPU registers.

When setting up the MPU, and if the MPU has previously been programmed, disable unused regions to prevent any previous region settings from affecting the new MPU setup.
### MPU configuration for a microcontroller

Usually, a microcontroller system has only a single processor and no caches. In such a system, program the MPU as follows:

<table>
<thead>
<tr>
<th>Memory region</th>
<th>TEX</th>
<th>C</th>
<th>B</th>
<th>S</th>
<th>Memory type and attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash memory</td>
<td>0b00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Normal memory, Non-shareable, write-through</td>
</tr>
<tr>
<td>Internal SRAM</td>
<td>0b00</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>Normal memory, Shareable, write-through</td>
</tr>
<tr>
<td>External SRAM</td>
<td>0b00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Normal memory, Shareable, write-back, write-allocate</td>
</tr>
<tr>
<td>Peripherals</td>
<td>0b00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Device memory, Shareable</td>
</tr>
</tbody>
</table>

In most microcontroller implementations, the shareability and cache policy attributes do not affect the system behavior. However, using these settings for the MPU regions can make the application code more portable. The values given are for typical situations. In special systems, such as multiprocessor designs or designs with a separate DMA engine, the shareability attribute might be important. In these cases see the recommendations of the memory device manufacturer.
4.6 Floating Point Unit (FPU)

This section describes the optional Floating Point Unit (FPU) in a Cortex-M4F device. The FPU implements the FPv4-SP floating-point extension.

The FPU fully supports single-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. It also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions.

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

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

Table 4-49 shows the floating-point system registers in the Cortex-M4F FPU.

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Type</th>
<th>Reset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xE000ED88</td>
<td>CPACR</td>
<td>RW</td>
<td>0x00000000</td>
<td>Coprocessor Access Control Register</td>
</tr>
<tr>
<td>0xE000EF34</td>
<td>FPCCR</td>
<td>RW</td>
<td>0xC0000000</td>
<td>Floating-point Context Control Register on page 4-49</td>
</tr>
<tr>
<td>0xE000EF38</td>
<td>FPCAR</td>
<td>RW</td>
<td>-</td>
<td>Floating-point Context Address Register on page 4-50</td>
</tr>
<tr>
<td>-</td>
<td>FPSCR</td>
<td>RW</td>
<td>-</td>
<td>Floating-point Status Control Register on page 4-50</td>
</tr>
<tr>
<td>0xE000EF3C</td>
<td>FPDSR</td>
<td>RW</td>
<td>0x00000000</td>
<td>Floating-point Default Status Control Register on page 4-52</td>
</tr>
</tbody>
</table>

The following sections describe the floating-point system registers whose implementation is specific to this processor.

4.6.1 Coprocessor Access Control Register

The CPACR register specifies the access privileges for coprocessors. See the register summary in Cortex-M4F floating-point system registers for its attributes. The bit assignments are:

<table>
<thead>
<tr>
<th>31 30 29 28</th>
<th>27 26 25 24</th>
<th>23 22 21 20</th>
<th>19 18 17 16</th>
<th>15 14 13 12</th>
<th>11 10 9 8</th>
<th>7 6 5 4</th>
<th>3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>CP11</td>
<td>CP10</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-50 CPACR register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2n+1:2n]</td>
<td>CPn</td>
<td>Access privileges for coprocessor n. The possible values of each field are:</td>
</tr>
<tr>
<td>for n values 10 and 11</td>
<td></td>
<td>0b00 = Access denied. Any attempted access generates a NOCP UsageFault.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b01 = Privileged access only. An unprivileged access generates a NOCP fault.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b10 = Reserved. The result of any access is unpredictable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b11 = Full access.</td>
</tr>
<tr>
<td>[19:0]</td>
<td>-</td>
<td>Reserved. Read as Zero, Write Ignore.</td>
</tr>
</tbody>
</table>
4.6.2 Floating-point Context Control Register

The FPCCR register sets or returns FPU control data. See the register summary in *Cortex-M4F floating-point system registers* on page 4-48 for its attributes. The bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ASPEN</td>
<td>Enables \texttt{CONTROL&lt;2&gt;} 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 \texttt{CONTROL&lt;2&gt;} setting on execution of a floating-point instruction. 1 = Enable \texttt{CONTROL&lt;2&gt;} setting on execution of a floating-point instruction.</td>
</tr>
<tr>
<td>30</td>
<td>LSPEN</td>
<td>0 = Disable automatic lazy state preservation for floating-point context. 1 = Enable automatic lazy state preservation for floating-point context.</td>
</tr>
<tr>
<td>29:9</td>
<td>Reserved</td>
<td>- Reserved.</td>
</tr>
<tr>
<td>8</td>
<td>MONRDY</td>
<td>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.</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
<td>- Reserved.</td>
</tr>
<tr>
<td>6</td>
<td>BFRDY</td>
<td>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.</td>
</tr>
<tr>
<td>5</td>
<td>MMRDY</td>
<td>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.</td>
</tr>
<tr>
<td>4</td>
<td>HFRDY</td>
<td>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.</td>
</tr>
<tr>
<td>3</td>
<td>THREAD</td>
<td>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.</td>
</tr>
</tbody>
</table>
4.6.3 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 Cortex-M4F floating-point system registers on page 4-48 for its attributes. The bit assignments are:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| [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. |

### Table 4-51 FPCCR register bit assignments (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:3]</td>
<td>ADDRESS</td>
<td>The location of the unpopulated floating-point register space allocated on an exception stack frame.</td>
</tr>
</tbody>
</table>

4.6.4 Floating-point Status Control Register

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>N</td>
<td>Condition code flags. Floating-point comparison operations update these flags.</td>
</tr>
<tr>
<td>[29]</td>
<td>C</td>
<td>Zero condition code flag.</td>
</tr>
<tr>
<td>[28]</td>
<td>V</td>
<td>Carry condition code flag.</td>
</tr>
<tr>
<td>[27]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>

### Table 4-52 FPCAR register bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:3]</td>
<td>ADDRESS</td>
<td>The location of the unpopulated floating-point register space allocated on an exception stack frame.</td>
</tr>
</tbody>
</table>

### Table 4-53 FPSCR bit assignments

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31]</td>
<td>N</td>
<td>Condition code flags. Floating-point comparison operations update these flags.</td>
</tr>
<tr>
<td>[29]</td>
<td>C</td>
<td>Zero condition code flag.</td>
</tr>
<tr>
<td>[28]</td>
<td>V</td>
<td>Carry condition code flag.</td>
</tr>
<tr>
<td>[27]</td>
<td>-</td>
<td>Reserved.</td>
</tr>
</tbody>
</table>
### Table 4-53 FPSCR bit assignments (continued)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[26]</td>
<td>AHP</td>
<td>Alternative half-precision control bit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: IEEE half-precision format selected.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Alternative half-precision format selected.</td>
</tr>
<tr>
<td>[25]</td>
<td>DN</td>
<td>Default NaN mode control bit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: NaN operands propagate through to the output of a floating-point operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Any operation involving one or more NaNs returns the Default NaN.</td>
</tr>
<tr>
<td>[24]</td>
<td>FZ</td>
<td>Flush-to-zero mode control bit:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0: Flush-to-zero mode disabled. Behavior of the floating-point system is fully compliant with the IEEE 754 standard.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1: Flush-to-zero mode enabled.</td>
</tr>
<tr>
<td>[23:22]</td>
<td>RMode</td>
<td>Rounding Mode control field. The encoding of this field is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b00: Round to Nearest (RN) mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b01: Round towards Plus Infinity (RP) mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b10: Round towards Minus Infinity (RM) mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b11: Round towards Zero (RZ) mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The specified rounding mode is used by almost all floating-point instructions.</td>
</tr>
<tr>
<td>[7]</td>
<td>IDC</td>
<td>Input Denormal cumulative exception bit, see bits [4:0].</td>
</tr>
<tr>
<td>[4]</td>
<td>IXC</td>
<td>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.</td>
</tr>
<tr>
<td>[0]</td>
<td>IOC</td>
<td>DZC: Division by Zero cumulative exception bit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOC: Invalid Operation cumulative exception bit.</td>
</tr>
</tbody>
</table>
4.6.5 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 *Cortex-M4F floating-point system registers* on page 4-48 for its attributes. The bit assignments are:

```
<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td>22</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>[31:27]</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>[26]</td>
<td>AHP</td>
<td>Default value for FPSCR.AHP</td>
</tr>
<tr>
<td>[25]</td>
<td>DN</td>
<td>Default value for FPSCR.DN</td>
</tr>
<tr>
<td>[24]</td>
<td>FZ</td>
<td>Default value for FPSCR.FZ</td>
</tr>
<tr>
<td>[23:22]</td>
<td>RMode</td>
<td>Default value for FPSCR.RMode</td>
</tr>
<tr>
<td>[21:0]</td>
<td>-</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

### 4.6.6 Enabling the FPU

The FPU is disabled from reset. You must enable it before you can use any floating-point instructions. Example 4-1 shows an example code sequence for enabling the FPU in both privileged and user modes. The processor must be in privileged mode to read from and write to the CPACR.

#### Example 4-1 Enabling the FPU

```assembly
; CPACR is located at address 0xE000ED88
LDR.W R0, =0xE000ED88 ; Read CPACR
LDR R1, [R0] ; Set bits 20-23 to enable CP10 and CP11 coprocessors
ORR R1, R1, #(0xF << 20) ; Write back the modified value to the CPACR
STR R1, [R0]; wait for store to complete
DSB ;reset pipeline now the FPU is enabled
ISB
```
Appendix A
Cortex-M4 Options

This appendix describes the configuration options for a Cortex-M4 processor implementation. It shows what features of a Cortex-M4 implementation are determined by the device manufacturer. It contains the following section:

•  *Cortex-M4 implementation options on page A-2.*
## A.1 Cortex-M4 implementation options

Table A-1 shows the Cortex-M4 implementation options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description, and affected documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inclusion of MPU</td>
<td>The implementer decides whether to include the Memory Protection Unit (MPU). See the Optional Memory Protection Unit on page 4-37.</td>
</tr>
<tr>
<td>Inclusion of FPU</td>
<td>Only the Cortex-M4F includes the Floating Point Unit (FPU). See:</td>
</tr>
<tr>
<td></td>
<td>• Floating-point instructions on page 3-126</td>
</tr>
<tr>
<td></td>
<td>• Interruptible-continuable instructions in Core registers on page 2-3</td>
</tr>
<tr>
<td></td>
<td>• The FPACTV bit in the CONTROL register</td>
</tr>
<tr>
<td></td>
<td>• Table 2-17 on page 2-28</td>
</tr>
<tr>
<td></td>
<td>• the MLSERR bit in the MemManage Fault Status Register (MMFSR)</td>
</tr>
<tr>
<td></td>
<td>• the LSPERR bit in the BusFault Status Register (BFSR).</td>
</tr>
<tr>
<td>Number of interrupts</td>
<td>The implementer decides how many interrupts the Cortex-M4 implementation supports Cortex-M4 implementation supports, in the range 1-240. This affects:</td>
</tr>
<tr>
<td></td>
<td>• The range of IRQ values in Table 2-5 on page 2-6</td>
</tr>
<tr>
<td></td>
<td>• Entries in the last row of Table 2-16 on page 2-22, particularly if only one interrupt is implemented.</td>
</tr>
<tr>
<td></td>
<td>• Exception handlers on page 2-23</td>
</tr>
<tr>
<td></td>
<td>• Figure 2-2 on page 2-24</td>
</tr>
<tr>
<td></td>
<td>• Nested Vectored Interrupt Controller on page 4-3.</td>
</tr>
<tr>
<td></td>
<td>• The number of implemented Nested Vectored Interrupt Controller (NVIC) registers in:</td>
</tr>
<tr>
<td></td>
<td>• Table 4-2 on page 4-3</td>
</tr>
<tr>
<td></td>
<td>• The appropriate register descriptions in sections Interrupt Set-enable Registers on page 4-4</td>
</tr>
<tr>
<td></td>
<td>• Interrupt Priority Registers on page 4-7.</td>
</tr>
<tr>
<td></td>
<td>Vector Table Offset Register on page 4-16, including the figure and Table 4-16 on page 4-16. See the</td>
</tr>
<tr>
<td></td>
<td>configuration information in the section for guidance on the required configuration.</td>
</tr>
<tr>
<td>Number of priority bits</td>
<td>The implementer decides how many priority bits are implemented in priority value fields, in the range 3-8. This affects The maximum priority level value in Nested Vectored Interrupt Controller on page 4-3.</td>
</tr>
<tr>
<td>Inclusion of the WIC</td>
<td>The implementer decides whether to include the Wakeup interrupt Controller (WIC), see The optional Wakeup Interrupt Controller on page 2-33.</td>
</tr>
<tr>
<td>Sleep mode power-saving</td>
<td>The implementer decides what sleep modes to implement, and the power-saving measures associated with any implemented mode, See Power management on page 2-32.</td>
</tr>
<tr>
<td></td>
<td>Sleep mode power saving might also affect the SysTick behavior, see SysTick usage hints and tips on page 4-36.</td>
</tr>
<tr>
<td>Register reset values</td>
<td>The implementer decides whether all registers in the register bank can be reset. This affects the reset values, see Table 2-2 on page 2-3.</td>
</tr>
<tr>
<td>Endianness</td>
<td>The implementer decides whether the memory system is little-endian or big-endian, see on page 2-10Data types on page 2-10 and Memory endianness on page 2-18.</td>
</tr>
<tr>
<td>Memory features</td>
<td>Some features of the memory system are implementation-specific. This means that the Memory model on page 2-12 cannot completely describe the memory map for a specific Cortex-M4 implementation.</td>
</tr>
<tr>
<td>Option</td>
<td>Description, and affected documentation</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Bit-banding</td>
<td>The implementer decides whether bit-banding is implemented., see <em>Optional bit-banding on page 2-16</em> and <em>Memory model on page 2-12</em>.</td>
</tr>
<tr>
<td>SysTick timer</td>
<td>The SYST_CALIB register is implementation-defined. This can affect:</td>
</tr>
<tr>
<td></td>
<td>• <em>SysTick Calibration Value Register on page 4-35</em></td>
</tr>
<tr>
<td></td>
<td>• The entry for SYST_CALIB in <em>Table 4-32 on page 4-33</em>.</td>
</tr>
</tbody>
</table>
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-M4 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.