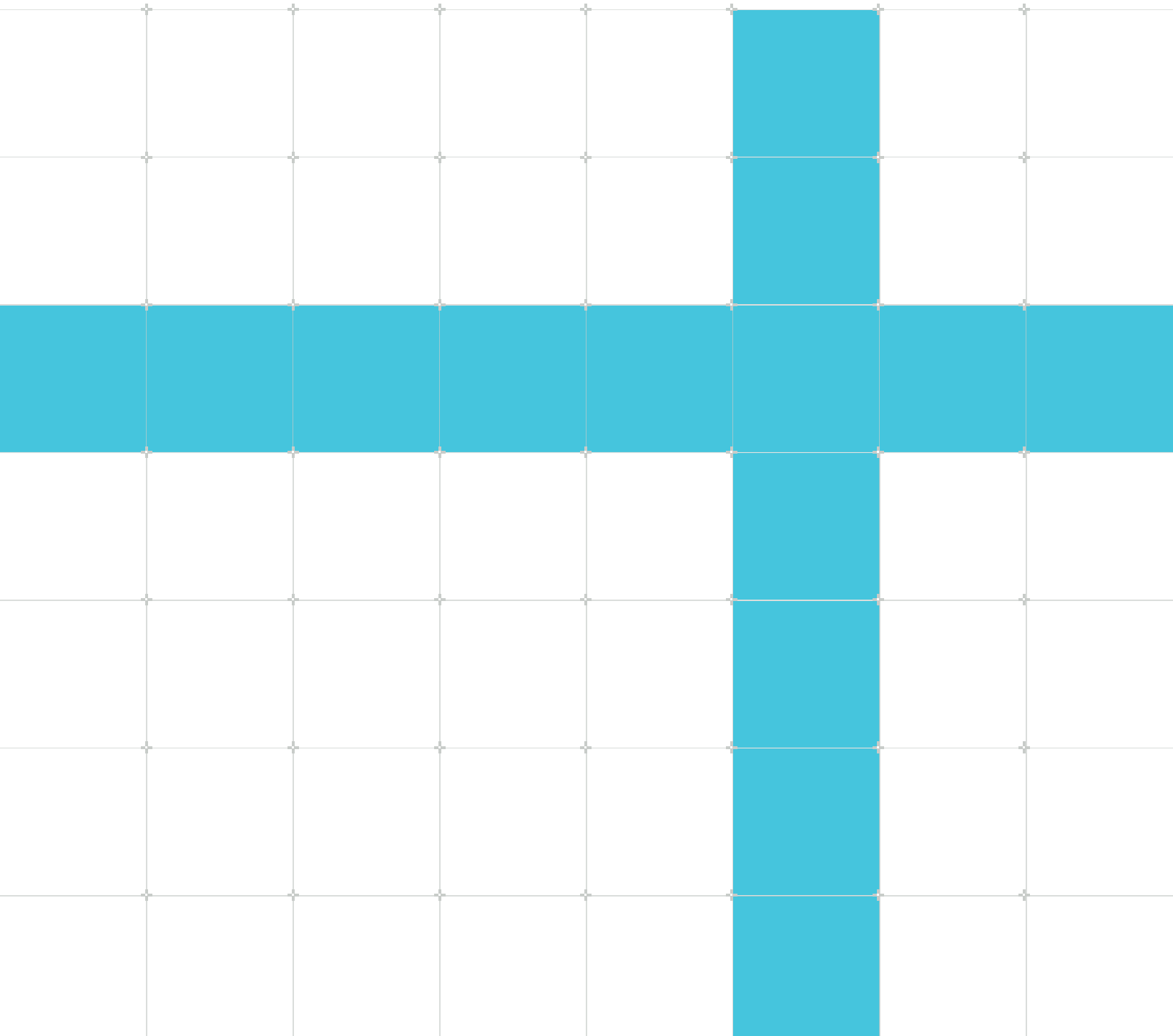
ARM Cortex-R Series (Armv7-R) Programmer's Guide

Version 1.0

Issue 01
den0042_0100_01_en



ARM Cortex-R Series (Armv7-R) Programmer's Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	3 April 2014	Non-Confidential	First release

Proprietary Notice

makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

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

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

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

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

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

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

Inclusive language commitment

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

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

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

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

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

See the Arm Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Citations.
bold	Interface elements, such as menu names. Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

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

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

1.2 Useful resources

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

Arm documents are available on developer.arm.com/documentation.

Confidential documents are only available to licensees, when logged in. Each document link in the tables below provides direct access to the online version of the document.

1.3 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.

- Technical Support.
- Arm® Glossary.

2. Introduction to Cortex-R

An embedded system can be defined as a piece of computer hardware running software designed to perform a specific task. This contrasts with what is generally considered a computer system, that is, one that runs a wide range of general purpose software running multiple tasks.

Embedded systems are commonly found in consumer, industrial, automotive, medical, commercial and military applications, with a controlling SoC designed for a single purpose ranging from digital watches to traffic lights, and larger, more complex systems. Aircraft contain advanced avionics such as inertial guidance systems and GPS receivers, both systems with considerable safety requirements. Automotive safety systems include Anti-lock Braking System (ABS), Electronic Stability Control (ESC/ESP), Traction Control (TCS) and automatic four-wheel drive.

In dealing with security, the embedded systems can also be self-sufficient and be able to deal with a failure of electrical and communication systems. Complexity can vary from single core devices, to multiple units, peripherals and networks.

There are many constraints on embedded systems that can make programming them more of a challenge than writing an application for a general-purpose PC.

Memory Footprint

In many systems, to minimize cost and power, memory size is limited. This forces you to consider the size of the program and how to reduce memory usage while it runs.

Power

In many embedded systems the power source is a battery. Programmers and hardware designers must take great care to minimize the total energy usage of the system.

Real-time behavior

A feature of certain systems is that there are time constraints to respond to external events. This might be a hard requirement or soft requirement. An example hard requirement is a car braking system because it must respond within a certain time consistently. An example soft requirement is an audio processing system because it must complete within a certain time. The ARM Real-time (R) profile defines an architecture aimed at systems that require deterministic timing and low interrupt latency.

A system is said to be real-time if the total correctness of an operation depends not only on its logical correctness, but also on the time in which it is performed. Real-time systems, and their deadlines, are classified by the consequence of missing a deadline:

Hard

Missing a deadline is a total system failure.

Firm

Infrequent deadline misses are tolerable, but can degrade the systems quality of service. The usefulness of a result is zero after its deadline.

Soft

The usefulness of a result degrades after its deadline, thereby degrading the system's Quality of Service.

The goal of a hard real-time system is to ensure that all deadlines are met, but for soft real-time systems the goal can be meeting a certain subset of deadlines to optimize an application-specific criteria. The particular subset of deadlines depends on the application, but some typical examples might include minimizing the lateness of tasks and maximizing the number of high priority tasks meeting their deadlines.

Hard real-time systems are used when it is vital that an event be reacted to within a strict deadline. Such guarantees are required of systems for which not reacting in a certain interval of time would cause great loss in some manner, especially damaging the surroundings physically or threatening human lives, although the strict definition is that missing the deadline constitutes failure of the system. For example, a car engine control systemis a hard real-time system because any delayed signal might cause engine failure or damage to the engine. Other examples include medical systems such as heart pacemakers and industrial process controllers. Hard real-time systems are typically found interacting at a low level with physical hardware, in embedded systems.

Cortex-R series cores are intended for use in deeply-embedded, real-time systems. providing high-performance computing solutions where reliability, high availability, fault tolerance, maintainability and real-time responses are required. This requires optimizing the core for performance while enabling time-critical code to execute in a timely manner. Cortex-R series processors include features to improve the determinism of the core for critical code.

For embedded applications requiring high performance combined with high reliability, Cortex-R series cores also provide several useful features, including both soft and hard error management, redundant dual-core systems using two cores in lock-step and Error Correcting Codes (ECC) on all external buses.

- The ARM Cortex-R4 processor is a mid-range real-time processor for use in deeply embedded systems.
- The ARM Cortex-R4F processor is a Cortex-R4 processor with a floating-point unit (FPU).
- The ARM Cortex-R5 processor is a high-performance real-time processor for use in embedded systems.
- The ARM Cortex-R5F processor is a Cortex-R5 processor with a floating-point unit (FPU).
- The ARM Cortex-R7 MPCore processor is a high performance real-time multi-core processor for use in a vast range of deeply embedded applications.

The Cortex-R series processors differ from both the Cortex-M and Cortex-A series processors. Cortex-R series processors typically offer much higher performance and can run at higher clock speeds than the Cortex-M series, while the Cortex-A series is intended for user-facing applications with complex software operating systems employing virtual memory management.

ARM Partners have developed families of devices using the Cortex-R4 processor with varying feature sets and levels of performance for products ranging from 3G USB modem sticks to automotive microcontrollers such as the TMS570 devices available from Texas Instruments. Infineon also has a Cortex-R4 processor-based medical device platform, MD8710. In all cases these devices are enabled by the specific capabilities of the Cortex-R4 processor, namely high computing performance, configurable features such as soft error handling, and the ability to respond deterministically to hard real-time events in an embedded system

Hard disk drives also continue to be one of the most demanding applications for embedded processors and the Cortex-R series has been adopted by many of the major manufacturers. For HDD and SSD mass storage systems the Cortex-R series processors provide a balance between processor performance, reliability and real-time response, along with ease of development and CoreSight debug to support current and future system chip architectures. Cortex-R series processors are also being widely adopted for Solid State Drives where performance and real time responses both to the host interface and the NAND devices is key to the system performance.

Mobile handsets are introducing high data rate wireless broadband to deliver feature-rich, audio, video and Internet services to users. Advanced multi-core SoCs use Cortex-R series processors for these tasks, complementing Cortex-A series processors for user applications. Low cost and power consumption continue to be key success criteria for mobile handset products. The real-time features of Cortex-R series processors are particularly suited to advanced mobile baseband applications and include support for high frequency interrupts along with fast and deterministic control of data transmission between cellular base-stations and mobile devices.

2.1 Determinism

Deeply-embedded, real-time systems require optimizing the processor for performance while enabling time-critical code to execute in a timely manner. Cortex-R processors include features to improve the predictability (determinism) of the processor for critical code. The memory subsystems of Cortex-R processors use Tightly Coupled Memory (TCM) located close to the processor core, meaning the core can have immediate access to memory rather than having to wait for external memory. This is described in Tightly Coupled Memory.

The Cortex-R5 and Cortex-R7 processors include ports specifically to provide access to time-critical peripherals. Having the dedicated ports means that accesses to these peripherals do not contend with lower priority memory accesses in the rest of the memory subsystem. This is described in Access to peripherals.

The Cortex-R processors also contain a number of features that provide deterministic timing and low interrupt latency for hard real-time applications. These features are collectively referred to as Low Latency Interrupt features. This is described in Low latency interrupts.

The Cortex-R7 processor has an out-of-order pipeline, in contrast to the in-order pipeline of the Cortex-R4 and Cortex-R5 processors. This means that the Cortex-R7 processor is more optimized towards performance than the Cortex-R4 and Cortex-R5 processors. The Cortex-R7 processor also includes features to help prioritize more critical code. These features are called Quality of Service (QoS) features and are described in Quality of Service.

3. ARM Architecture and Processors

Periodically, new versions of the architecture are announced by ARM. These add new features or make changes to existing behaviors. Such changes are almost always backwards compatible, meaning that user code that ran on older versions of the architecture will continue to run correctly on new versions. Of course, code written to take advantage of new features will not run on older processors that lack these features.

In all versions of the architecture, some system features and behaviors are implementation-defined. For example, the architecture does not define cache sizes or cycle timings for individual instructions. These are determined by the individual processor and SoC.

The ARMv7 architecture also has the concept of profiles. These are variants of the architecture describing processors targeting different markets and uses.

The profiles are:

A

The Application profile defines an architecture aimed at high performance processors and supports a virtual memory system using a Memory Management Unit (MMU). It is capable of running fully featured operating systems. It has support for the ARM and Thumb instruction sets.

R

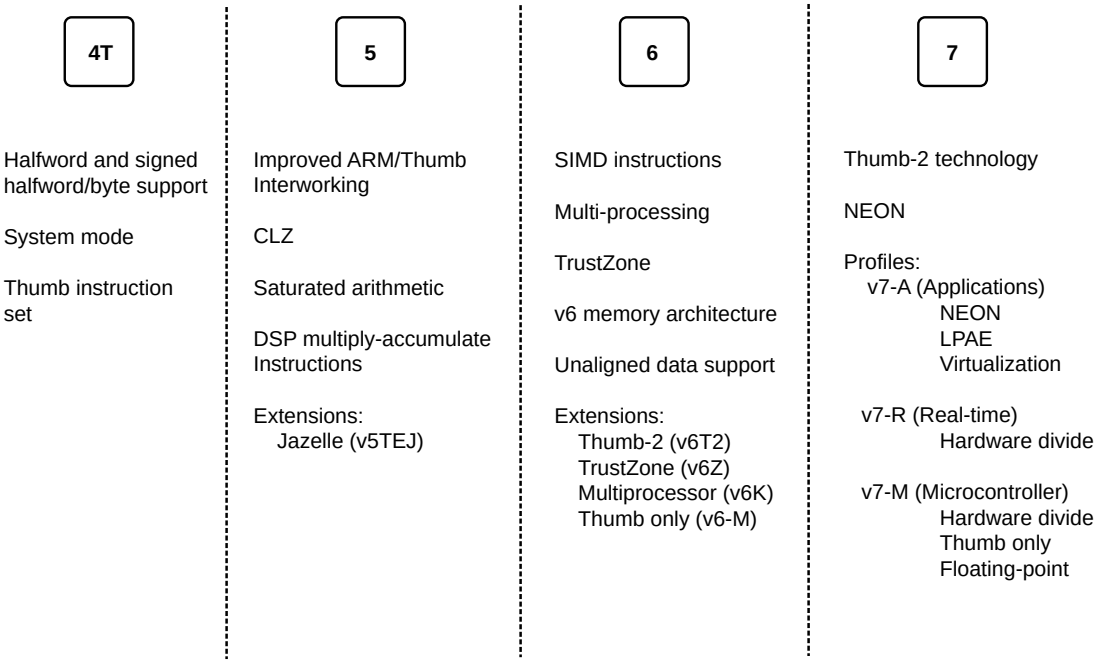
The Real-time profile defines an architecture aimed at systems that require deterministic timing and low interrupt latency. These systems use a Memory Protection Unit (MPU) to protect the memory regions. These systems do not support a virtual memory system and MMU.

M

The Microcontroller profile defines an architecture aimed at low cost systems, where low-latency interrupt processing is vital. It uses a different exception handling model to the other profiles and supports only a variant of the Thumb instruction set.

Figure 3-1: Architecture history on page 20 shows the development of the architecture over time, illustrating additions to the architecture at each new version. Almost all architecture changes are backwards compatible, meaning software written for the ARMv4T architecture can be used on ARMv7 processors.

Figure 3-1: Architecture history

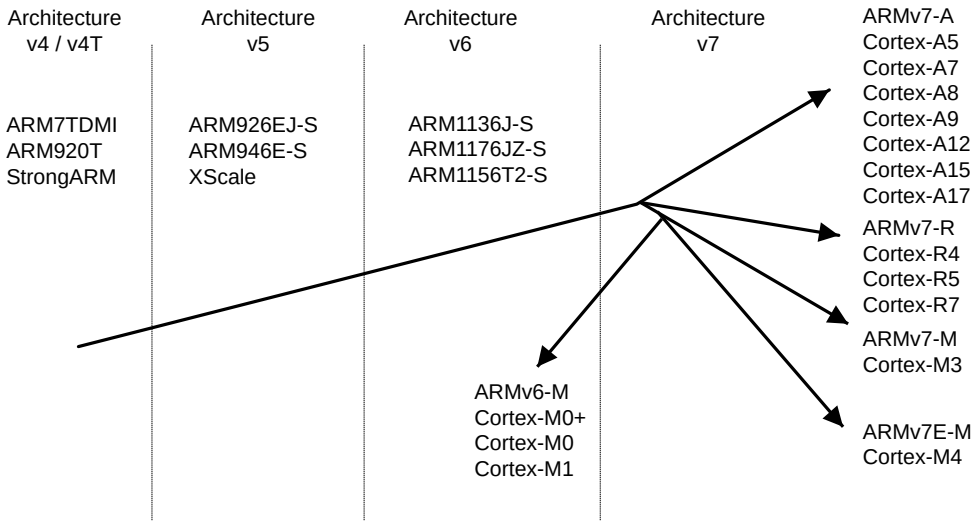


3.1 Processor properties

For many years, ARM adopted a sequential numbering system for processors with ARM9 following ARM8, which came after ARM7. Various numbers and letters were appended to the base family to denote different variants. For example, the ARM7TDMI processor has T for Thumb, D for Debug, M for a fast multiplier and I for EmbeddedICE.

For the ARMv7 architecture, ARM adopted the brand name Cortex for its processors, with a supplementary letter indicating which of the three profiles, A, R, or M, the processor supports. Figure 3-2: Architecture and processors on page 21 shows how different versions of the architecture correspond to different processor implementations. The figure is not comprehensive and does not include all architecture versions or processor implementations.

Figure 3-2: Architecture and processors



This section briefly looks at some ARM processors and identifies which processor implements which architecture version. Cortex-R series processors takes a slightly more detailed look at some of the individual processors that implement architecture version ARMv7-R. The terminology in this chapter might be unfamiliar to the first-time user of ARM processors.

Table 3-1: Cortex processors and architecture versions on page 21 lists the architecture version implemented by the Cortex family of processors.

Table 3-1: Cortex processors and architecture versions

v7-R (Real-time)	v7-A (Application)	v6-M/v7-M (Microcontroller)
Cortex-R4	Cortex-A5 (Single/MP)	Cortex-M0+ (ARMv6-M)
Cortex-R5	Cortex-A7 (MP)	Cortex-M0 (ARMv6-M)
Cortex-R7	Cortex-A8 (Single)	Cortex-M1 (ARMv6-M)
	Cortex-A9 (Single/MP)	Cortex-M3 (ARMv7-M)
	Cortex-A12 (MP)	Cortex-M4(F) (ARMv7E-M)
	Cortex-A15 (MP)	
	Cortex-A17 (MP)	

Table 3-2: Properties of Cortex-R series processors on page 21 compares the properties of Cortex-R series processors.

Table 3-2: Properties of Cortex-R series processors

Property	Cortex-R4	Cortex-R5	Cortex-R7
Release date	May 2006	August 2010	March 2012
Typical clock speed	600 MHz on 40 nm	600 MHz on 40 nm	1 GHz on 28 nm
Execution order	In-order	In-order	Out-of-order
Cores	1	1 to 2 (in AMP mode)	1 to 2

Property	Cortex-R4	Cortex-R5	Cortex-R7
Peak integer throughput	2.45 DMIPS/MHz	2.45 DMIPS/MHz	2.50 DMIPS/MHz
VFP architecture	VFPv3	VFPv3	VFPv3
Half precision extension	No	No	Yes
Hardware integer divide	Yes	Yes	Yes
Fused Multiply Accumulate	No	No	No
Pipeline stages	8	8	11
Instruction decode per cycle	Partial dual issue	Partial dual issue	2 (Superscalar)
Return stack entries	4	4	8
Floating Point Unit	Optional	Optional	Optional
AMBA interface	64-bit AXI 3	64-bit AXI 3	64-bit AXI 3
Generic Interrupt Controller	Not included	Not included	Included
Branch predictor entries	256	256	Configurable
Indirect predictor	No BTAC	No BTAC	256, 512, 1024, 2048, or 4096 BTAC entries
Trace	Optional ETM Separate macrocell	Optional ETM, Separate macrocell	Optional ETM
Cache	I-Cache and D-Cache	I-Cache and D-Cache	I-Cache and D-Cache
Pipeline	8 stage dual issue	8 stage dual issue	11 stage superscalar with Out-of-order execution

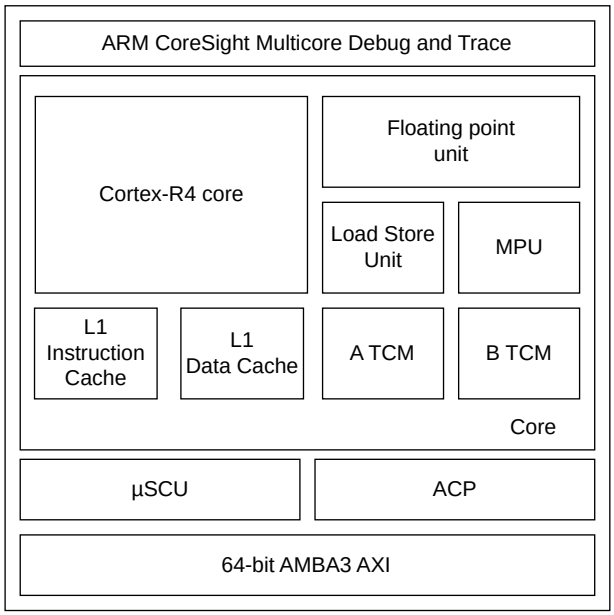
3.2 Cortex-R series processors

This section takes a closer look at each of the processors that implement the ARMv7-R architecture. It gives only a general description in each case. For more specific information on each processor, see Table 3-2: Properties of Cortex-R series processors on page 21.

3.3 The Cortex-R4 processor

The ARM Cortex-R4 processor is designed for use in embedded, real-time systems. It implements the ARMv7-R architecture, and includes Thumb-2 technology for optimum code density and processing throughput, but can also use ARM instructions. It is for use in high-volume, deeply embedded System-on-Chip applications, for example, hard disk drive controllers, wireless baseband processors, consumer products and electronic control units for automotive systems.

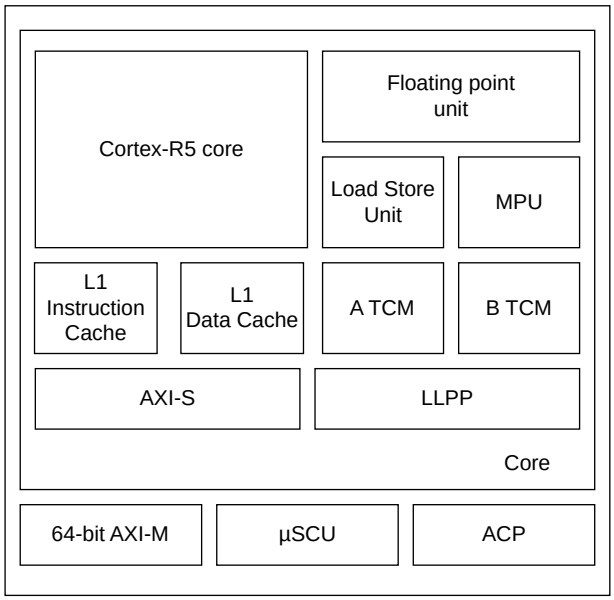
Figure 3-3: Cortex-R4 core block diagram



3.4 The Cortex-R5 processor

The Cortex-R5 processor is designed for use in embedded, real-time systems. It implements the ARMv7-R architecture, and includes Thumb-2 technology for optimum code density and processing throughput, but can also use ARM instructions. The pipeline has a single Arithmetic Logic Unit, but implements limited dual-issuing of instructions for efficient utilization of other resources such as the register file. A hardware Accelerator Coherency Port (ACP) is available to reduce the requirement for slow software cache maintenance operations when sharing memory with other masters.

Figure 3-4: Cortex-R5 core block diagram

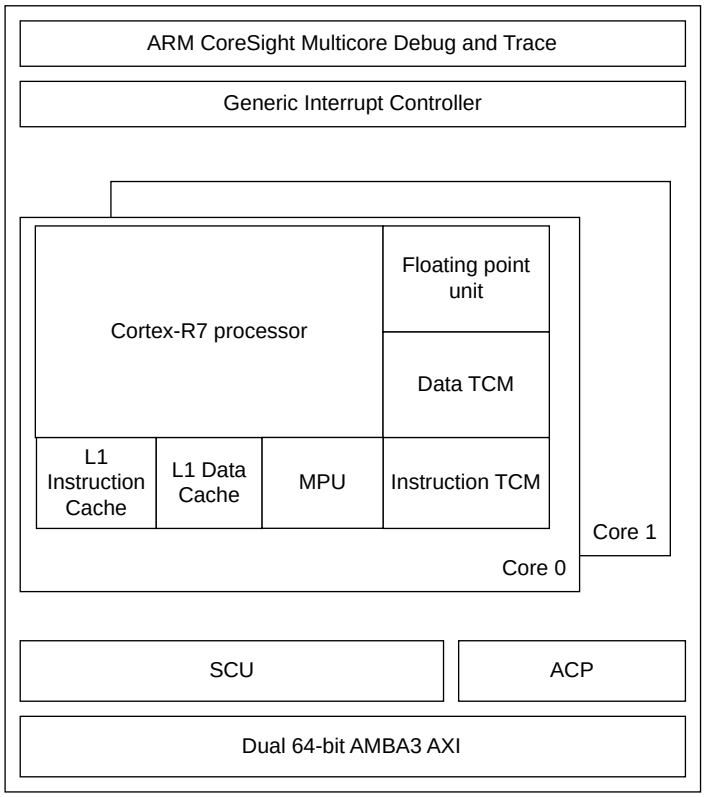


3.5 The Cortex-R7 processor

The ARM Cortex-R7 MPCore processor is a highly configurable processor for use in embedded real-time systems. It consists of one or two Cortex-R7 cores in a single cluster. It implements the ARMv7-R architecture, and includes Thumb-2 technology for optimum code density and processing throughput.

The Cortex-R7 processor provides higher levels of performance to the Cortex-R series of processors through the introduction of out-of-order instruction execution and dynamic register re-naming, combined with improved branch prediction, superscalar execution capability and faster hardware support for divide, DSP and floating point functions.

Figure 3-5: Cortex-R7 processor block diagram



The Cortex-R7 processor has the following features:

- Flexible Multi-Processor Core (MPCore) configurations.
- Lock-step configuration with redundant core.
- Symmetric Multi-Processing (SMP).
- Asymmetric Multi-Processing (AMP).
- Integrated Generic Interrupt Controller (GIC), Snoop Control Unit (SCU), and timers.
- Quality of Service (QoS) features.
- Full coherency support for SMP.
- Hardware accelerated data cache operation with Tag RAM copies in SCU.
- Twin core and I/O coherency.
- Dedicated Low Latency Peripheral and Memory Ports for hard real-time work.
- Flexible and configurable Floating Point Unit (FPU).
- CoreSight SoC Debug and Trace.
- Optional Embedded Trace Macrocell ETMv4.

3.6 Development platforms

There are off-the-shelf ARM processor based platforms that are suitable for students and hobbyists:

MCBTMS570

The MCBTMS570 Development Kit is an Cortex-R4 based board set that can be used use to generate and test application programs for the Texas Instruments TMS570x microcontroller. The MCBTMS570 Board contains all the hardware components required in a single-chip TMS570x system.

The TMS570LS series is a high performance automotive grade microcontroller family that has been certified for use in IEC 61508 SIL3 safety systems and integrates the Cortex-R4F processor running at 1.6DMIPS/MHz, and has configurations that can run up to 160 MHz providing more than 250 DMIPS. The TMS570LS series also provides different Flash (1MB or 2MB) and data SRAM (128KB or 160KB) options with single bit error correction and double bit error detection.

The TMS570LS series is an ideal solution for high performance real time control applications with safety critical requirements.

Hercules

The Hercules safety microcontroller platform consists of two Cortex R4-based microcontroller families: TMS570 and RM4x. Designed specifically for IEC 61508 and ISO 26262 safety critical applications, the Hercules platform provides advanced integrated safety features while delivering scalable performance, connectivity, and memory options.

3.7 Operating systems for Cortex-R processors

Code running on Cortex-R processors might be running as bare-metal applications without an operating system, or use a Real-Time Operating System (RTOS). Real-time operating systems include:

- Integrity from Green Hills
- Nucleus from Mentor Graphics
- Extended T-Kernel from eSOL.

Many other operating systems are available. Cortex-R processors can also run suitably modified versions of Linux, configured to run without the use of a full MMU.

4. ARM Processor modes and Registers

The ARMv7-R architecture is a modal architecture. There are six privileged modes and a non-privileged user mode. Privilege, in this case, is the ability to perform certain tasks that cannot be done from User (unprivileged) mode. In User mode, there are limitations on operations that affect overall system configuration. For example, only privileged modes can change the operating mode. Modes are associated with exception events, described in Exceptions and Interrupts.

Table 4-1: ARM processor modes

Mode	Function	Privilege
User (USR)	Mode in which most programs and applications run	Unprivileged
FIQ	Entered on an FIQ interrupt exception	Privileged
IRQ	Entered on an IRQ interrupt exception	Privileged
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Privileged
Abort (ABT)	Entered on a memory access exception	Privileged
Undef (UND)	Entered when an undefined instruction is executed	Privileged
System (SYS)	Runs tasks that require a privileged processor mode, shares the register view with User mode.	Privileged

The introduction of the TrustZone Security Extensions in the ARMv6Z architecture created two security states (Secure and Non-secure) that are independent of Privilege and processor mode. The ARMv7-A architecture Virtualization Extensions also add a hypervisor mode (Hyp) in addition to the existing privileged modes. While the ARMv7-R architecture profile does not implement either the Security Extensions or Virtualization Extensions, their existence in the ARMv7 architecture has led to the redefinition of privileged (unprivileged or privileged) modes of operation in terms of Privilege levels.

Every memory access has an access privilege, that is either unprivileged or privileged.

The ARMv7-R architecture profile defines the following privilege levels:

PL0

The privilege level of application software, that executes in User mode. Therefore, software executed in User mode is described as unprivileged software. This software cannot access some features of the system. In particular, it cannot change many of the configuration settings.

Software executing at PL0 makes only unprivileged memory accesses.

PL1

Software execution in all modes other than User mode is at PL1. Normally, operating system software executes at PL1.

The PL1 modes refers to all the modes other than User mode.

We can therefore redefine the processor modes in terms of privilege levels.

Table 4-2: ARM processor modes and privilege levels

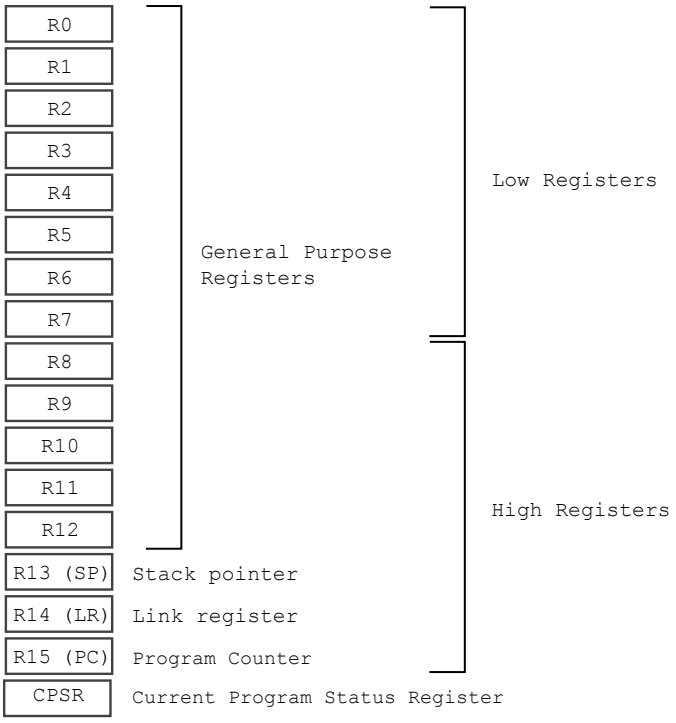
Mode	Function	Privilege level
User (USR)	Mode in which most programs and applications run	PL0
FIQ	Entered on an FIQ interrupt exception	PL1
IRQ	Entered on an IRQ interrupt exception	PL1
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	PL1
Abort (ABT)	Entered on a memory access exception	PL1
Undef (UND)	Entered when an undefined instruction is executed	PL1
System (SYS)	Runs tasks that require a privileged processor mode, shares the register view with User mode.	PL1

The current processor mode and execution state is contained in the Current Program Status Register (CPSR). Changing processor state and modes can be either explicit by privileged software, or as a result of taking exceptions.

4.1 Registers

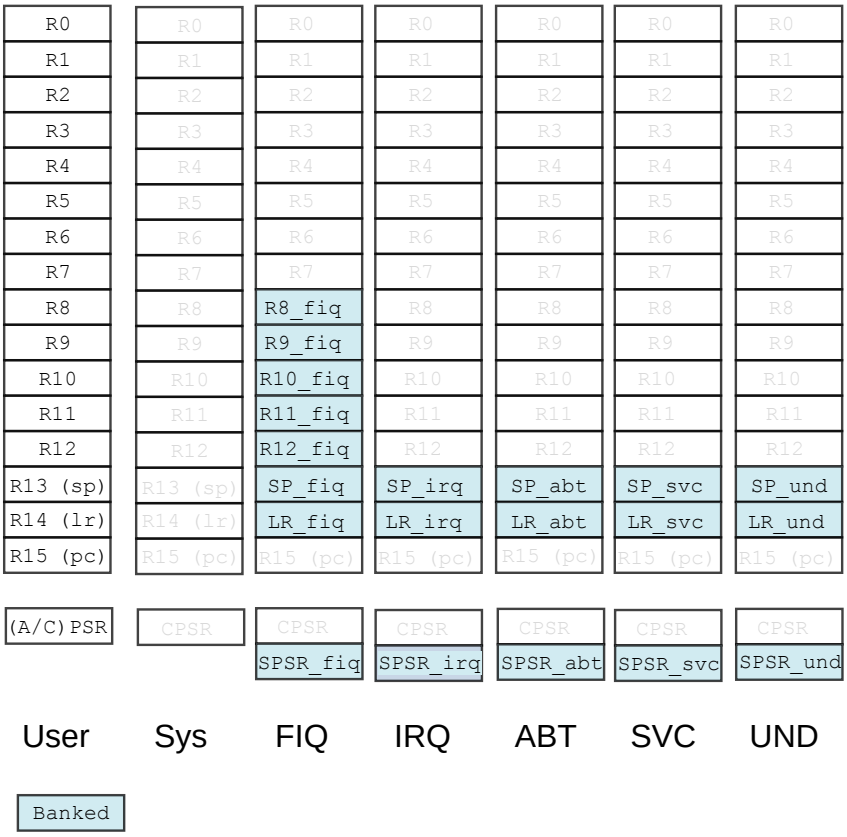
The ARMv7-R architecture provides sixteen 32-bit general purpose registers (R0-R15) for software use. Fifteen of them (R0-R14) can be used for general purpose data storage, while R15 is the program counter whose value is altered as the core executes instructions. An explicit write to R15 by software will alter program flow. Software can also access the CPSR and, in privileged modes a saved copy of the CPSR from the previously executed mode, called the Saved Program Status Register (SPSR).

Figure 4-1: Programmer visible registers for user code



Although software can access the registers, depending on the mode the software is executing in and the register being accessed, a register might correspond to a different physical storage location. This is called banking, the shaded registers in Figure 4-2: The ARMv7-R register set on page 29 are banked. They use physically distinct storage and are usually accessible only when a process is executing in that particular mode.

Figure 4-2: The ARMv7-R register set



In all modes, 'Low Registers' and R15 share the same physical storage location. Figure 4-2: The ARMv7-R register set on page 29 shows that some 'High Registers' are banked for certain modes. For example, R8 to R12 are banked for FIQ mode, that is, accesses to them go to a different physical storage location. For all modes other than User and System modes, R13, R14 and the SPSRs are banked.


Note

The stack pointer, R13, can be used as a general-purpose register only in ARM code but not in Thumb code.

In the case of banked registers, software does not normally specify the instance of the banked register. The instance of the banked register is implied by the mode from which the access is made. For example, a program executing in User mode that specifies R13 accesses R13_usr. A program executing in SVC mode that specifies R13 accesses R13_svc.

R14 (the Link Register) holds the return address from a subroutine entered when you use the branch with link (BL) instruction. It can be used as a general purpose register when it is not supporting returns from subroutines. R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are used similarly to hold the return address when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

R15 is the program counter and holds the current program address (actually, it always points eight bytes ahead of the current instruction in ARM state and four bytes ahead of the current instruction in Thumb state, a legacy of the three stage pipeline of the original ARM1 processor). In ARM state, bits[1:0] of R15 always read as zero. In Thumb state, bit[0] of R15 always reads as zero.

The reset values of R0-R14 are unpredictable. SP, the stack pointer, must be initialized, for each mode, by boot code before making use of the stack. The AAPCS or AEABI specifies how software should use the general purpose registers to interoperate between different toolchains or programming languages.

4.1.1 Program Status Registers

At any given moment, you have access to 16 registers (R0-R15) and the Current Program Status Register (CPSR). In User mode, a restricted form of the CPSR called the Application Program Status Register (APSR) is accessed instead.

The Current Program Status Register (CPSR) is used to store:

- The APSR flags.
- The current processor mode.
- Interrupt disable flags.
- The current processor state, that is, ARM, Thumb, ThumbEE, or Jazelle.
- The endianness.
- Execution state bits for the IT block.

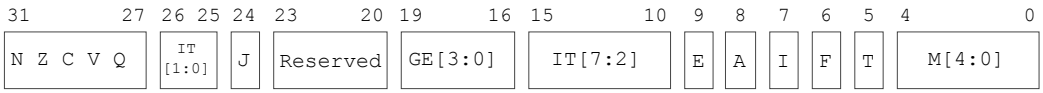
The Program Status Registers (PSRs) form an additional set of banked registers. Each exception mode has its own Saved Program Status Register (SPSR) where a copy of the pre-exception CPSR is stored automatically when an exception occurs. These are not accessible from User mode.

Application programmers must use the APSR to access the parts of the CPSR that can be changed in unprivileged mode. The APSR must be used only to access the N, Z, C, V, Q, and GE[3:0] bits. These bits are not normally accessed directly, but instead set by condition code setting instructions and tested by instructions that are executed conditionally.

For example `CMP R0, R1` compares the values of R0 and R1 and sets the zero flag (Z) if R0 and R1 are equal.

The CPSR contains condition code flags, the current mode bits, and other information as Figure 4-3: CPSR bits on page 31 shows.

Figure 4-3: CPSR bits



The individual bits represent the following:

N

Negative result from ALU.

Z

Zero result from ALU.

C

ALU operation Carry out.

V

ALU operation oVerflowed.

Q

Cumulative saturation (also described as sticky bit).

J

Indicates whether the processor is in Jazelle state.

GE[3:0]

Used by SIMD instructions.

IT[7:0], bits [15:10, 26:25]

If-Then conditional execution of Thumb instruction groups.

E

Controls endianness for load and store operations.

A

Value of 1 disables asynchronous aborts.

I

Value of 1 disables IRQ.

F

Value of 1 disables FIQ.

T

Indicates whether the processor is in Thumb state.

M[4:0]

Specifies the processor mode (see Table 4-2: ARM processor modes and privilege levels on page 28).

The processor can change between modes using instructions that directly write to the CPSR mode bits. More commonly, the processor changes mode automatically as a result of exception events. In User mode, you cannot manipulate bits [4:0] that control the processor mode or the A, I and F bits that govern the exceptions to be enabled or disabled.

4.1.2 Coprocessor 15

CP15, the System Control coprocessor, is (despite the name coprocessor) an integral part of the processor and provides control of many features. It can contain up to 16 primary registers, each of size 32 bits. However access to CP15 is privilege controlled and not all registers are available in User (unprivileged) mode.

The CP15 register access instructions specify the required primary register, with the other fields in the instruction used to define the access more precisely and increase the number of physical 32-bit registers in CP15. The 16 primary registers in CP15 are named c0 to c15, but are often referred to by a series of letters.

For example, the CP15 System Control Register is named CP15.SCTLR. Table 3.3 summarizes the function of some of the more relevant registers used in Cortex-R series processors. We will consider some of these in more detail when we look at the operation of the cache and MPU.

Table 4-3: CP15 Register summary

Register	Description
Main ID Register (MIDR)	Gives identification information for the processor, including implementer code and device ID.
Multiprocessor Affinity Register (MPIDR)	Provides a way to uniquely identify individual processors within a multi-processor system.
CP15 c1 System Control registers	
System Control Register (SCTLR)	The main processor control register, see System Control Register (SCTLR).
Auxiliary Control Register (ACTLR)	Implementation defined. Implementation specific additional control and configuration options.
Coprocessor Access Control Register (CPACR)	Controls access to all coprocessors except CP14 and CP15.
CP15 c5 and c6, memory system registers	
Data Fault Status Register (DFSR)	Gives status information about the last data fault (see Exceptions and Interrupts).
Instruction Fault Status Register (IFSR)	Gives status information about the last instruction fault (see Exceptions and Interrupts).
Data Fault Address Register (DFAR)	Gives the address of the access that caused the most recent precise data abort (see Exceptions and Interrupts).
Instruction Fault Address Register (IFAR)	Gives the address of the access that caused the most recent precise prefetch abort (see Exceptions and Interrupts).
MPU Region Base Address Register	Describes the base address of the region specified by the Memory Region Number Register. (See c6, MPU Region Base Address Register)

Register	Description
MPU Region Size and Enable Register	Specifies the size of the region specified by the Memory Region Number Register.Identifies the address ranges that are used for a particular region.Enables or disables the region, and its sub-regions, specified by the Memory Region Number Register. (See c6, MPU Region Size and Enable Register)
MPU Region Access Control Register	Holds the region attributes and access permissions for the region specified by the Memory Region Number Register. (See c6, MPU Region Access Control Register)
MPU Region Number Register	Multiple registers with one register for each memory region implemented. The value contained determines which of the multiple registers is accessed.
CP15 c7, cache maintenance and other functions	
Cache and branch predictor maintenance functions	See Caches.
Data and instruction barrier operations	See Memory Ordering.
CP15 c9, performance monitors	
CP15 c13, process, context and thread ID registers	
Context ID Register (CONTEXTIDR)	Holds a process identification value for the running process. It is available in Privileged mode only.
Software thread ID registers	Provide locations to store the IDs of software threads and processes of the operating system.
CP15 c15, implementation defined registers	
Configuration Base Address Register (CBAR)	Provides a base address for the GIC and Local timer type peripherals.

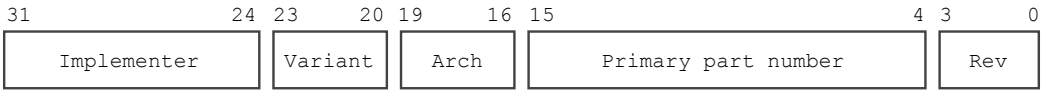
All system architecture functions are controlled by reading or writing a general purpose processor register (Rt) from or to a set of registers (CRn) located within Coprocessor 15. The Op1, Op2, and CRm fields of the instruction can also be used to select registers or operations. The format is as follows:

```
MRC p15, Op1, Rt, CRn, CRm, Op2 ; read a CP15 register into an ARM register

MCR p15, Op1, Rt, CRn, CRm, Op2 ; write a CP15 register from an ARM register
```

We will not go through each of the various CP15 registers in detail, as this would duplicate reference information that can readily be obtained from the ARM Architecture Reference Manual or product documentation. We will look at two examples, the read-only Main ID Register (MIDR), the format of which is shown in Figure 4-4: Main ID Register format on page 34 and the System Control Register (SCTLR).

Figure 4-4: Main ID Register format



In a privileged mode, we can read this register, using the instruction:

```
MRC p15, 0, <Rt>, c0, c0, 0
```

The result, placed in register `Rt`, tells software the processor it is running on. For a Cortex-R series processor conforming to the ARMv7-R architecture the results will be as follows:

Bits [31:24] - implementer, is `0x41` for an ARM designed processor.

Bits [23:20] - variant, shows the revision number of the processor.

Bits [19:16] - architecture, is `0xF` for ARM architecture v7.

Bits [15:4] - part number (for example `0xC15` for the Cortex-R5 processor).

Bits [3:0] - revision, shows the patch revision number of the processor.

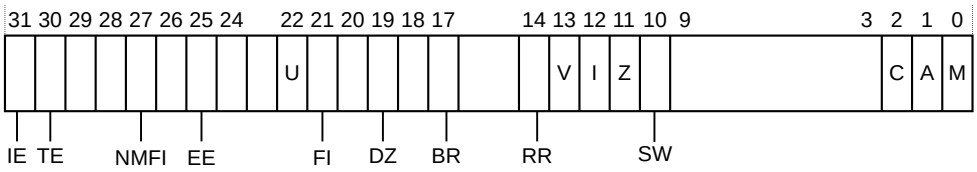
4.1.3 System Control Register (SCTLR)

The System Control Register (SCTLR) can be accessed using CP15. It:

- Controls standard memory.
- Controls system facilities.
- Provides status information for functions implemented in the core.

The SCTLR is only accessible from PL1 (privileged mode).

Figure 4-5: Simplified view of the System Control Register



The individual bits represent the following:

IE

Instruction Endianness (not on Cortex-R7 processor). This controls the endianness of instructions issued to the processor:

- Value 0 for little-endian.

- Value 1 for Big-endian.

TE

Thumb exception enable. This controls whether exceptions are taken in ARM or Thumb state:

- Value 0 for ARM state
- Value 1 for Thumb state.

NMFI

Non-maskable FIQ (NMFI) support.

EE

Exception endianness. This defines the value of the CPSR.E bit on entry to an exception:

- Value 0 for Little-endian
- Value 1 for Big-endian.

U

Not on Cortex-R7 processor. Indicates use of the alignment model.

FI

Fast interrupts configuration enable.

- Value 0 to enable all performance features.
- Value 1 to enable low interrupt latency. Disables some features.

DZ

Divide by zero fault bit.

- Value 0. Divide by zero returns zero, and no exception is taken.
- Value 1. Divide by zero cause as Undefined Instruction exception.

BR

Background region bit. See Implementing a Protected Memory System with Regions.

- Value 0. Any access to an address that is not mapped to an MPU region generates a Background fault memory abort.
- Value 1. The default memory map is used as a background region.

RR

Not on Cortex-R7 processor. Round Robin bit for cache implementation policy.

- Value 0. Normal replacement strategy, for example, random replacement.
- Value 1. Predictable strategy, for example, round-robin replacement.

V

This bit selects the base address of the exception vector table.

- Value 0 for low exception vectors, base address `0x00000000`.
- Value 1 for high exception vectors, base address `0xFFFF0000`.

I

Instruction cache enable bit.

- Value 0 to disable instruction caches.
- Value 1 to enable instruction caches.

Z

Branch prediction enable bit.

- Value 0 to disable branch prediction.
- Value 1 to enable branch prediction.

SW

SWP/SWPB enable bit.

- Value 0 to disable branch prediction.
- Value 1 to enable branch prediction.

ARM recommends that an ARMv7-R implementation does not include support for these instructions. When use of this bit is supported, at reset, this bit disables `SWP` and `SWPB`. This means that operating systems have to choose to use `SWP` and `SWPB`.

C

Cache enable bit.

- Value 0 to disable data and unified caches.
- Value 1 to enable data and unified caches.

A

Alignment check enable bit.

- Value 0 to disable alignment fault checking.
- Value 1 to enable alignment fault checking.

M

Enable the MPU. See Implementing a Protected Memory System with Regions.

- Value 0 disables MPU.
- Value 1 enables MPU.

Part of the boot code sequence will typically set the Z bit in the CP15:SCTLR, System Control Register, that enables branch prediction. A code sequence to do this is shown below:

```
MRC     p15, 0, r0, c1, c0, 0   ; Read System Control Register configuration data
ORR     r0, r0, #(1 << 2)       ; Set C bit
ORR     r0, r0, #(1 << 12)      ; Set I bit
ORR     r0, r0, #(1 << 11)      ; Set Z bit
MCR     p15, 0, r0, c1, c0, 0   ; Write System Control Register configuration data
```

A similar form of this code can be found in the example in Boot Code.

Data endianness is controlled by the CPSR.E and SCTLR.EE bits. The CPSR.E bit can be changed using the `SETEND` instruction. It also provides instructions for converting the format of data held in registers. These include the `REV` and `REV16` instructions.

5. Introduction to Assembly Language

Assembly language is a low-level programming language. There is in general a one-to-one relationship between assembly language instructions (mnemonics) and the actual binary opcode executed by the processor.

Many programmers writing at the application level have little need to code in assembly language. However, knowledge of assembly code can be useful in cases where highly optimized code is required, when writing compilers, or where low level use of features not directly available in C is required. It might be required for portions of boot code, device drivers or when performing OS development. Finally, it can be useful to be able to read assembly code when debugging C, and particularly, to understand the mapping between assembly instructions and C statements.

Programmers seeking a more detailed description of ARM assembly language can also refer to the ARM Compiler Toolchain Assembler Reference or the ARM Architecture Reference Manual.

5.1 Comparison with other assembly languages

An ARM processor is typically a Reduced Instruction Set Computer (RISC) processor. Complex Instruction Set Computer (CISC) processors, like the x86, have a rich instruction set capable of doing complex things with a single instruction. Such processors often have significant amounts of internal logic that decode machine instructions to sequences of internal operations (microcode). RISC architectures, in contrast, have a smaller number of more general purpose instructions, that might be executed with significantly fewer transistors, making the silicon cheaper and more power efficient. Like other RISC architectures, ARM processors have a large number of general-purpose registers and many instructions execute in a single cycle. It has simple addressing modes, where all load/store addresses can be determined from register contents and instruction fields.

The ARMv7 architecture has basic data processing instructions that permit them to perform arithmetic operations (such as `ADD`) and logical bit manipulation (such as `AND`). They also have branch instructions that transfer program execution from one part of the program to another, to support loops and conditional statements. The architecture also has instructions to read and write external memory.

The ARM instruction set is generally considered to be simple, logical, and efficient. It has features not found in other processors, while at the same time lacking operations found in others. For example, it cannot perform data processing operations directly on memory. To increment a value in a memory location, the value must be loaded to an ARM register, the register incremented and a third instruction is required to write the updated value back to memory. The Instruction Set Architecture (ISA) includes instructions that combine a shift with an arithmetic or logical operation, auto-increment and auto-decrement addressing modes for optimized program loops, Load, and Store Multiple instructions that enable efficient stack and heap operations, plus block copying capability and conditional execution of almost all instructions.

Like the x86 processors (but unlike the 68K processor), ARM instructions typically have a two or three operand format, with the first operand in most cases specifying the destination for the result.

Load multiple and store instructions are an exception to this rule. The 68K, by contrast, places the destination as the last operand. For ARM instructions, there are generally no restrictions on which registers can be used as operands. The following examples give a flavor of the differences between the different assembly languages.

```
x86:   add     eax, #100
68K:   ADD     #100, D0
ARM:   add     r0, r0, 100
```

```
x86:   mov     eax, DWORD PTR [ebx]
68K:   MOVE.L  (A0), D0
ARM:   ldr     r0, [r1]
```

5.2 The ARM instruction sets

The ARMv7 architecture is a 32-bit processor architecture. It is a load/store architecture, meaning that data-processing instructions operate on values in general purpose registers. Only load and store instructions access memory. General purpose registers are also 32 bits. Throughout this book, when we refer to a word, we mean 32 bits. A doubleword is therefore 64 bits and a halfword is 16 bits wide.

Though the ARMv7 architecture is a 32-bit architecture, individual processor implementations do not necessarily have 32-bit width for all blocks and interconnections. For example, it is possible to have 64-bit, or wider paths for instruction fetches or data accesses.

Most ARM processors support a number of different instruction sets, while some (for example, the Cortex-M3 processor) do not in fact execute the original ARM instruction set. There are at least two instruction sets that ARM processors can use.

ARM (32-bit instructions)

This is the original ARM instruction set.

Thumb

The Thumb instruction set was first added in the ARM7TDMI processor and contained only 16-bit instructions, that gave much smaller programs (memory footprint can be a major concern in smaller embedded systems) at the cost of some performance. Recent processors, including those in the Cortex-R series, support Thumb-2 technology, that extends the Thumb instruction set to provide a mix of 16-bit and 32-bit instructions. This gives the best of both worlds, performance similar to that of ARM, with code size similar to that of Thumb. Because of its size and performance advantages, it is increasingly common for all code to be compiled or assembled to take advantage of Thumb-2 technology.

Older ARM processors often contained code that was compiled for ARM state and code that was compiled for Thumb state. ARM code, with 32-bit instructions, was more powerful and required fewer instructions to perform a particular task and so might be preferred for performance critical parts of the system. It was also used for exception handler code, because exceptions could not be handled in Thumb state on ARM7 or ARM9 Series processors.

Thumb code, using 16-bit instructions, requires more instructions to carry out the same task, when compared with ARM code. Thumb code can typically encode smaller constant values within instructions and has shorter relative branches. See Branches. The available range for relative branches is approximately ±32MB for ARM instructions and ±16MB for the Thumb-2 extension. Thumb is also limited where only 16-bit instructions are used, with conditional branches having a range of ±256 Bytes and unconditional relative branches being limited to ±2048 bytes.

However, because Thumb instructions are only half of the size, programs are typically a third smaller than their ARM code equivalent. Thumb instructions are therefore used when code density is important, and to reduce system memory requirements. Thumb code can also outperform ARM when the processor is directly connected to a narrow (16-bit) memory, without the benefit of cache. One Thumb instruction can be fetched on each cycle, whereas each 32-bit ARM instruction requires two clock cycles per fetch.

When executing a Thumb instruction, the PC reads as the address of the current instruction plus 4. The only 16-bit Thumb instructions that can directly modify the PC are certain encodings of `MOV` and `ADD`. The value written to the PC is forced to be halfword-aligned by ignoring its least significant bit, treating that bit as being 0.

In ARMCC, the option `--thumb` or `-arm` (the default) enables selection of the instruction set used for compilation. A program can branch between these two instruction sets at run-time.

The currently used instruction set is indicated by the CPSR T bit and the core is said to be in ARM state (T = 0) or Thumb state (T = 1). Code has to be explicitly compiled or assembled to one state or the other. An explicit instruction is used to change between instruction sets. Calling functions that are compiled for a different state is known as interworking. We'll take a more detailed look at this in Interworking.

For Thumb assembly code, there is often a choice of 16-bit and 32-bit instruction encodings, with the 16-bit versions being generated by default. The `.W` (32-bit) and `.N` (16-bit) width specifiers can be used to force a particular encoding (if such an encoding exists), for example:

```
BCS.W   label   ; forces 32-bit instruction even for a short branch
B.N     label   ; faults if label out of range for 16-bit instruction
```

5.2.1 Thumb-2

Despite continued rumours to the contrary, there is no such thing as a Thumb-2 instruction set. Thumb-2 technology was introduced in ARMv6T2, and is required in ARMv7. This technology extends the original 16-bit Thumb instruction set to include 32-bit instructions. The range of 32-bit Thumb instructions included in ARMv6T2 permits Thumb code to achieve performance similar to ARM code, with code density better than that of the purely 16-bit Thumb code.

5.3 Introduction to the GNU Assembler

The GNU Assembler, part of the GNU tools, is used to convert assembly language source code into binary object files. The assembler is extensively documented in the GNU Assembler Manual, which can be found online at http://sourceware.org/binutils/docs/as/index.html or (if you have GNU tools installed on your system) in the `gnutools/doc` sub-directory. GNU Assembler documentation is also available in the `/gcc-doc/` package on Ubuntu.

What follows is a brief description, intended to highlight differences in syntax between the GNU Assembler and standard ARM assembly language, and provide enough information to let you get started with the tools.

The names of GNU tool components have prefixes indicating the target options selected, including operating system. An example would be `arm-none-eabi-gcc`, that might be used for bare metal systems using the ARM EABI.

5.3.1 Invoking the GNU Assembler

You can assemble the contents of an ARM assembly language source file by running the `arm-none-eabi-as` program.

```
arm-none-eabi-as -g -o filename.o filename.s
```

The option `-g` requests the assembler to include debug information in the output file.

When all of your source files have been assembled into binary object files (with the extension `.o`), you use the GNU Linker to create the final executable in ELF format.

This is done by executing:

```
arm-none-eabi-ld -o filename.elf filename.o
```

For more complex programs, where there are many separate source files, it is more common to use a utility like `make` to control the build process.

You can use the debugger provided by either `arm-none-eabi-gdb` or `arm-none-eabi-insight` to run the executable files on your host machine, as an alternative to a real target processor.

5.3.2 GNU Assembler syntax

The GNU Assembler can target many different processor architectures and is not ARM-specific. This means that its syntax is somewhat different from other ARM assemblers, such as the ARM

toolchain. The GNU Assembler uses the same syntax for all of the many processor architectures that it supports.

Assembly language source files consist of a sequence of statements, one per line.

Each statement has three optional parts, ordered as follows:

```
label: instruction @ comment
```

A label lets you identify the address of this instruction. This can then be used as a target for branch instructions or for load and store instructions. A label can be a letter followed (optionally) by a sequence of alphanumeric characters, followed by a colon.

The instruction can be either an ARM assembly instruction, or an assembler directive. These are pseudo-instructions that tell the assembler itself to do something. These are required, amongst other things, to control sections and alignment, or create data.

Everything on the line after the `@` symbol is treated as a comment and ignored (unless it is inside a string). C style comment delimiters `/*` and `*/` can also be used.

At link time an entry point can be specified on the command line if one has not been explicitly provided in the source code.

5.3.3 Sections

An executable program with code will have at least one section, that by convention is called `.text`. Data can be included in a `.data` section.

Directives with these names enable you to specify which of the two sections should hold what follows in the source file. Executable code should appear in a `.text` section and read or write data in the `.data` section. Also read-only constants can appear in a `.rodata` section. Zero initialized data appears in `.bss`. The Block Started by Symbol (bss) segment defines the space for uninitialized static data.

5.3.4 Assembler directives

This is a key area of difference between GNU tools and other assemblers.

All assembler directives begin with a period "." A full list of these is described in the GNU documentation. Here, we give a subset of commonly used directives.

`.align`

This causes the assembler to pad the binary with bytes of zero value, in data sections, or **NOP** instructions in code, ensuring the next location is on a word boundary. The `.align n` gives 2^n alignment on ARM processors.

`.ascii "string"`

Insert the string literal into the object file exactly as specified, without a NUL character to terminate. Multiple strings can be specified using commas as separators.

`.asciiz`

Does the same as `.ascii`, but this time additionally followed by a NUL character (a byte with the value 0 (zero)).

`.byte expression, .hword expression, .word expression`

Inserts a byte, halfword, or word value into the object file. Multiple values can be specified using commas as separators. The synonyms `.2byte` and `.4byte` can also be used.

`.data`

Causes the following statements to be placed in the data section of the final executable.

`.end`

Marks the end of this source code file. The assembler does not process anything in the file after this point.

`.equ symbol, expression`

Sets the value of symbol to expression. The "=" symbol and `.set` have the same effect.

`.extern symbol`

Indicates that symbol is defined in another source code file.

`.globalsymbol`

Tells the assembler that symbol is to be made globally visible to other source files and to the linker.

`.include "filename"`

Inserts the contents of filename into the current source file and is typically used to include header files containing shared definitions.

`.text`

This switches the destination of following statements into the text section of the final output object file. Assembly instructions must always be in the text section.

For reference, Table 5-1: Comparison of syntax on page 43 shows common assembler directives alongside GNU and ARM tools. Not all directives are listed, and in some cases there is not a 100% correspondence between them.

Table 5-1: Comparison of syntax

GNU Assembler	armasm	Description
`@`	`;`	Comment
`#&`	`#0x`	An immediate hex value
`.if`	`IFDEF, IF`	Conditional (not 100% equivalent)
`.else`	`ELSE`	
`.elseif`	`ELSEIF`	
`.endif`	`ENDIF`	
`.ltorg`	`LTORG`	

GNU Assembler	armasm	Description
|	:OR:	OR
&	:AND:	AND
<<	:SHL:	Shift Left
>>	:SHR:	Shift Right
.macro	MACRO	Start macro definition
.endm	ENDM	End macro definition
.include	INCLUDE	GNU Assembler requires "filename"
.word	DCD	A data word
.short	DCW	
.long	DCD	
.byte	DCB	
.req	RN	
.global	IMPORT, EXPORT	
.equ	EQU	

5.3.5 Expressions

Assembly instructions and assembler directives often require an integer operand. In the assembler, this is represented as an expression to be evaluated. Typically, this is an integer number specified in decimal, hexadecimal (with a `0x` prefix) or binary (with a `0b` prefix) or as an ASCII character surrounded by single quotes.

In addition, standard mathematical and logical expressions can be evaluated by the assembler to generate a constant value. These can utilize labels and other pre-defined values. These expressions produce either absolute or relative values. Absolute values are position-independent and constant. Relative values are specified relative to some linker-defined address, determined when the executable image is produced - such as target addresses for branches.

5.3.6 GNU tools naming conventions

Registers are named in GCC as follows:

- General registers: R0 - R15.
- Stack pointer register: SP (R13).
- Frame pointer register: FP (R11).
- Link register: LR (R14).
- Program counter: PC (R15).
- Program Status Register flags: xPSR, xPSR_all, xPSR_f, xPSR_x, xPSR_ctl, xPSR_fs, xPSR_fx, xPSR_f, xPSR_cs, xPSR_cf, xPSR_cx (where x = C current or S saved). See Program Status Registers.

5.4 ARM tools assembly language

The Unified Assembly Language (UAL) format now used by ARM tools enables the same canonical syntax to be used for both ARM and Thumb instruction sets. The assembler syntax of ARM tools is not identical to that used by the GNU Assembler, particularly for preprocessing and pseudo-instructions that do not map directly to opcodes. In the next chapter, we will look at the individual assembly language instructions in a little more detail. Before doing that, we take a look at the basic syntax used to specify instructions and registers. Assembly language examples in this book use both UAL and GNU Assembly syntax.

UAL gives the ability to write assembler code that can be assembled to run on all ARM processors. In the past, it was necessary to write code explicitly for ARM or Thumb state. Using UAL the same code can be assembled for different instruction sets at the time of assembly, not at the time the code is written. This can be either through the use of command line switches or inline directives. Legacy code will still assemble correctly. It is worth noting that GNU Assembler now supports UAL through use of the .syntax directive, though it might not be identical syntax to the ARM tools assembler.

5.4.1 ARM assembler syntax

ARM assembler source files consist of a sequence of statements, one per line.

Each statement has three optional parts, ordered as follows:

```
label instruction ; comment
```

A label lets you identify the address of this instruction. This can then be used as a target for branch instructions or for load and store instructions.

The instruction can be either an assembly instruction, or an assembler directive. These are pseudo-instructions that tell the assembler itself to do something. These are required, amongst other things, to control sections and alignment, or create data.

Everything on the line after the `;` symbol is treated as a comment and ignored (unless it is inside a string). C style comment delimiters `/*` and `*/` can also be used.

5.4.2 Labels

A label is required to start in the first character of a line. If the line does not have a label, a space or tab delimiter is required to start the line. If there is a label, the assembler makes the label equal to the address in the object file of the corresponding instruction. Labels can then be used as the target for branches or for loads and stores.

```
Loop        MUL R5, R5, R1
            SUBS R1, R1, #1
```

```
            BNE Loop
```

In this example, `Loop` is a label and the conditional branch instruction (`BNE Loop`) is assembled in a way that makes the offset encoded in the branch instruction point to the address of the `MUL` instruction that is associated with the label `Loop`.

5.4.3 Directives

Most lines will normally have an assembly language instruction, to be converted by the tool into its binary equivalent, or a directive which tells the assembler to do something. It can also be a pseudo-instruction, that is, one that is converted into one or more real instructions by the assembler. We'll look at the actual instructions available in hardware in Unified Assembly Language Instructions and focus mainly on the assembler directives here. These perform a wide range of tasks. They can be used to place code or data at a particular address in memory, create references to other programs and so forth.

For example, the Define Constant (`DCD`, `DCB`, `DCW`) directives let us place data into a piece of code. This can be expressed numerically (in decimal, hex, binary) or as ASCII characters. It can be a single item or a comma separated list. `DCB` is for byte sized data, `DCD` can be used for word sized data, and `DCW` for half-word sized data items.

For example, you might have:

```
MESSAGE DCB "Hello World!",0
```

This will produce a series of bytes corresponding to the ASCII characters in the string, with a 0 termination. `MESSAGE` is a label that you can use to get the address of this data. Similarly, you might have data items expressed in hex:

```
Masks DCD 0x100, 0x80, 0x40, 0x20, 0x10
```

The `EQU` directive lets us assign names to address or data values. For example:

```
CtrlD EQU 4

TUBE EQU 0x30000000
```

You can then use these labels in other instructions, as parts of expressions to be evaluated. `EQU` does not actually cause anything to be placed in the program executable - it merely sets a name to a value, for use in other instructions, in the symbol table for the assembler. It is convenient to use such names to make code easier to read, but also so that if you change the address or value of something in a piece of code, you only have to modify the original definition, rather than having to change all of the references to it individually. It is usual to group `EQU` definitions together, often at the start of a program or function, or in separate include files.

The AREA pseudo-instruction is used to tell the assembler about how to group together code or data into logical sections for later placement by the linker. For example, exception vectors might have to be placed at a fixed address. The assembler keeps track of where each instruction or piece of data is located in memory. The AREA directive can be used to modify that location.

The ALIGN directive lets you align the current location to a specified boundary. It usually does this by padding (where necessary) with zeros or NOP instructions, although it is also possible to specify a pad value with the directive. The default behavior is to set the current location to the next word (four byte) boundary, but larger boundary sizes and offsets from that boundary can also be specified. This can be required to meet alignment requirements of certain instructions (for example LDRD and STRD doubleword memory transfers), or to align with cache boundaries. As with the `.align` directive on GNU Assembler, the `ALIGN n` directive gives 2^n alignment on ARM processors.

END is used to denote the end of the assembly language source program. Failure to use the END directive will result in an error being returned.

INCLUDE tells the assembler to include the contents of another file into the current file. Include files can be used as an easy mechanism for sharing definitions between related files.

5.5 Interworking

When the processor executes ARM instructions, it is said to be operating in ARM state. When it is operating in Thumb state, it is executing Thumb instructions. A processor in a particular state can only sensibly execute instructions from that instruction set. You must make sure that the processor does not receive instructions of the wrong instruction set.

Each instruction set includes instructions to change processor state. ARM and Thumb code can be mixed, if the code conforms to the requirements of the ARM and Thumb Procedure Call Standards. Compiler generated code will always do so, but assembly language programmers must take care to follow the specified rules.

Selection of processor state is controlled by the T bit in the CPSR. See Figure 4-3: CPSR bits on page 31. When T is 1, the processor is in Thumb state. When T is 0, the processor is in ARM state. However, when the T bit is modified, it is also necessary to flush the instruction pipeline (to avoid problems with instructions being decoded in one state and then executed in another). Special instructions are used to accomplish this. These are BX (Branch with eXchange) and BLX (Branch and Link with eXchange). LDR of PC and POP/LDM of PC also have this behavior. In addition to changing the processor state with these instructions, assembly programmers must also use the appropriate directive to tell the assembler to generate code for the appropriate state.

The BX or BLX instruction branches to an address contained in the specified register, or an offset specified in the opcode. The value of bit [0] of the branch target address determines whether execution continues in ARM state or Thumb state. This only applies to the forms of BX/BLX which take a register. The PC-relative forms cannot generate an address with lsb anything other than zero so will always change state regardless. Both ARM (aligned to a word boundary) and Thumb (aligned to a halfword boundary) instructions do not use bit [0] to form an address. This bit can therefore

safely be used to provide the additional information about whether the `BX` or `BLX` instruction should change the state to ARM (address bit [0] = 0) or Thumb (address bit [0] = 1). The `BL label` can be turned into a `BLX label` as appropriate at link time if the instruction set of the caller is different from the instruction set of `label`, assuming that it is unconditional.

A typical use of these instructions is when a call from one function to another is made using the `BL` or `BLX` instruction, and a return from that function is made using the `BX LR` instruction. Alternatively, you can have a non-leaf function, that pushes the link register onto the stack on entry and pops the stored link register from the stack into the program counter, on exit. Here, instead of using the `BX LR` instruction to return, you instead have a memory load. Memory load instructions that modify the PC might also change the processor state depending on the value of bit [0] of the loaded address.

5.6 Identifying assembly code

When faced with a piece of assembly language source code, it can be useful to be able to determine which instruction set is used and which kind of assembler it is targeted at.

Older ARM Assembly language code can have three (or even four) operand instructions present (for example, `ADD R0, R1, R2`) or conditional execution of non-branch instructions (for example, `ADDNE R0, R0, #1`). Filename extensions will typically be `.s` or `.S`.

Code targeted for the newer UAL, will contain the directive `.syntax unified` but will otherwise appear similar to traditional ARM Assembly language. The pound (or hash) symbol `#` can be omitted in front of immediate operands. Conditional instruction sequences must be preceded immediately by the `IT` instruction described in Unified Assembly Language Instructions. Such code assembles either to fixed-size 32-bit (ARM) instructions, or mixed-size (16-bit and 32-bit) Thumb instructions, depending on the presence of the directives `.thumb` or `.arm`.

You can, on occasion, encounter code written in 16-bit Thumb assembly language. This can contain directives such as `.code 16`, `.thumb` or `.thumb_func` but will not specify `.syntax unified`. It uses two operands for most instructions, although `ADD` and `SUB` can sometimes have three. Only branches can be executed conditionally.

All GCC inline assembler, for example, `.c`, `.h`, `.cpp`, `.cxx`, and `.c++` code can be built for Thumb or ARM, depending on GCC configuration and command-line switches (`-marm` or `-mthumb`).

6. Unified Assembly Language Instructions

This chapter is a general introduction to Unified Assembly Language. It does not provide detailed coverage of every instruction.

Instructions can broadly be placed in one of a number of classes:

- Data operations (ALU operations such as `ADD`).
- Memory operations (load and stores to memory).
- Branches (for loops, goto, conditional code and other program flow control).
- DSP (operations on packed data, saturated mathematics and other special instructions, targeting codecs).
- Miscellaneous (coprocessor, debug, mode changes and so forth).

We'll take a brief look at each of those in turn. Before we do that, let us examine capabilities that are common to different instruction classes.

6.1 Instruction set basics

ARMv7-R architecture added support for hardware divide, the `UDIV` and `SDIV` instructions, which were unsupported on other architecture profiles. The behavior of data processing instructions that write to the PC on Cortex-R cores is also different. Previously data processing instructions could not cause a state change, except when returning from an exception. In ARMv7-R any data processing instruction that writes to the PC can change the state, based on bit [0] of the address. There are however, still a number of features common to all parts of the instruction set.

6.1.1 Constant and immediate values

ARM or Thumb assembly language instructions have a length of only 16 or 32 bits. This presents something of a problem. It means that you cannot encode an arbitrary 32-bit value within the opcode.

In the ARM instruction set, as opcode bits are used to specify condition codes, the instruction itself and the registers to be used, only 12 bits are available to specify an immediate value. We have to be somewhat creative in how these 12 bits are used. Rather than enabling a constant of size -2048 to +2047 to be specified, instead the 12 bits are divided into an 8-bit constant and 4-bit rotate value. The rotate value enables the 8-bit constant value to be rotated right by a number of places from 0 to 30 in steps of 2, that is, 0, 2, 4, 6, 8...

So, you can have immediate values like `0x23` or `0xFF`. You can produce other useful immediate values, for example, addresses of peripherals or blocks of memory. As an example, `0x23000000` can be produced by expressing it as `0x23 ROR 8`. But many other constants, like `0x3FF`, cannot be produced within a single instruction. For these values, you must either construct them in multiple instructions, or load them from memory. Programmers do not typically concern themselves with

this, except where the assembler gives an error complaining about an invalid constant. Instead, you can use assembly language pseudo-instructions to do whatever is necessary to generate the required constant

Constant values encoded in an instruction can be one of the following in Thumb:

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

where `XY` is a hexadecimal number in the range `0x00` to `0xFF`.

The `MOVW` instruction (move wide), will move a 16-bit constant into a register, while zeroing the top 16 bits of the target register. `MOVT` (move top) will move a 16-bit constant into the top half of a given register, without changing the bottom 16 bits. This permits a `MOV32` pseudo-instruction to construct any 32-bit constant. The assembler provides some more help here. The prefixes `:upper16:` and `:lower16:` enable you to extract the corresponding half from a 32-bit constant:

```
MOVW R0, #:lower16:label
MOVT R0, #:upper16:label
```

Although this requires two instructions, it does not require any extra space to store the constant, and there is no requirement to read a data item from memory.

You can also use pseudo-instructions such as `LDR Rn, =<constant>` or `LDR Rn, =label`. (This was the only option for older processors that lacked MOVW and MOVT.) The assembler will then use the best sequence to generate the constant in the specified register (one of `MOV`, `MVN` or an `LDR` from a literal pool). A literal pool is an area of constant data held within the code section, typically after the end of a function and before the start of another. If it is necessary to manually control literal pool placement, this can be done with an assembler directive - `LTORG` for armasm, or `.ltorg` when using GNU tools. The register loaded could be the program counter, would cause a branch.

This can be useful for absolute addressing or for references outside the current section; obviously this will result in position-dependent code. The value of the constant can be determined either by the assembler, or by the linker.

ARM tools also provides the related pseudo-instruction `ADR Rn, =label`. This uses a PC-relative `ADD` or `SUB`, to place the address of the label into the specified register, using a single instruction. If the address is too far away to be generated this way, the `ADRL` pseudo-instruction is used. This requires two instructions, that gives a better range. This can be used to generate addresses for position-independent code, but only within the same code section.

6.1.2 Conditional execution

A feature of the ARM instruction set is that nearly all instructions are conditional. On most other architectures, only branches or jumps can be executed conditionally. This can be useful in avoiding conditional branches in small `if/then/else` constructs or for compound comparisons.

As an example of this, consider code to find the smaller of two values, in registers R0 and R1 and place the result in R2. This is shown in the following example. The suffix `LT` indicates that the instruction should be executed only if the most recent flag-setting instruction returned less than; `GE` means greater than or equal.

```
@ Code using branches
  CMP     R0, R1
  BLT     .Lsmaller   @ if R0<R1 jump over
  MOV     R2, R1      @ R1 is less than or equal to R0
  B       .Lend       @ finish
.Lsmaller:
  MOV     R2, R0      @ R0 is less than R1
.Lend:
```

Now look at the same code written using conditional `MOV` instructions, rather than branches:

```
  CMP      R0, R1
  MOVGE    R2, R1  @ R1 is less than or equal to R1
  MOVLT    R2, R0  @ R0 is less than R1
```

The latter piece of code is both smaller and, on older ARM processors, is faster to execute. However, this code can actually be slower on some processors where inter-instruction dependencies could cause longer stalls than a branch, and branch prediction can reduce, or potentially eliminate the cost of branches.

As a reminder, this style of programming relies on the fact that status flags can be set optionally on some instructions. If the `MOVGE` instruction automatically set the flags, the program might not work correctly. Load and Store instructions never set the flags. For data processing operations, however, you have a choice. By default, flags are preserved during such instructions. If the instruction is suffixed with an `S` (for example, `MOVS` rather than `MOV`), the instruction will set the flags. The `S` suffix is not required, or permitted, for the explicit comparison instructions. The flags can also be set manually, by using the dedicated `PSR` manipulation instruction (`MSR`). Some instructions set the Carry flag (C) based on the carry from the ALU and others based on the barrel shifter carry (that shifts a data word by a specified number of bits in one clock cycle).

Thumb code has a somewhat different mechanism for conditional execution. Branches can be executed conditionally. Instructions can also be conditionally executed by using the Compare and

Branch on Zero (CBZ) and Compare and Branch on Non-Zero (CBNZ) instructions. These compare the value of a register against zero and branch on the result.

Thumb-2 technology also introduced the If-Then (IT) instruction, providing conditional execution for up to four consecutive instructions. The conditions might all be identical, or some might be the inverse of the others. Instructions within an IT block must also specify the condition code to be applied.

IT is a 16-bit instruction that enables nearly all Thumb instructions to be conditionally executed, depending on the value of the ALU flags, using the condition code suffix. The syntax of the instruction is IT{x{y{z}}} where x, y and z specify the condition switch for the optional instructions in the IT block, either Then (T) or Else (E), for example, ITTET.

```
ITT   EQ
SUBEQ r1, r1, #1
ADDEQ r0, r0, #60
```

Typically, IT instructions are auto-generated by the assembler, rather than being hand-coded. 16-bit instructions that normally change the condition code flags, will not do so inside an IT block, except for CMP, CMN and TST whose only action is to set flags. There are some restrictions on which instructions can be used within an IT block. Exceptions can occur within IT blocks, the current if-then status is stored in the CPSR and so is copied into the SPSR on exception entry, so that when the exception returns, the execution of the IT block resumes correctly.

Certain instructions always set the flags and have no other effect. These are CMP, CMN, TST and TEQ, that are analogous to SUBS, ADDS, ANDS and EORS but with the result of the ALU calculation being used only to update the flags and not being placed in a register.

Table 6-1: Condition code suffixes on page 52 lists the 15 condition codes that can be attached to most instructions.

Table 6-1: Condition code suffixes

Sign	Suffix	Meaning	Flags
	EQ	Equal	Z = 1
	NE	Not equal	Z = 0
	CS	Carry set (identical to HS)	C = 1
	CC	Carry clear (identical to LO)	C = 0
	MI	Minus or negative result	N = 1
	PL	Positive or zero result	N = 0
	VS	Overflow	V = 1
	VC	Now overflow	V = 0
	AL	Always. This is the default	•
Unsigned	HI	Higher	C = 1 AND Z = 0
Unsigned	HS	Higher or same	C = 1
Unsigned	LS	Lower or same	C = 0 OR Z = 1

Sign	Suffix	Meaning	Flags
Unsigned	LO	Lower (identical to CC)	C = 0
Signed	GT	Greater than	Z = 0 AND N = V
Signed	GE	Greater than or equal	N = V
Signed	LE	Less than or equal	Z = 1 OR N != V
Signed	LT	Less than	N != V

6.1.3 Status flags and condition codes

The ARM processor has a Current Program Status Register (CPSR) that contains four status flags, (Z)ero, (N)egative, (C)arry and o(V)erflow.

Table 6-2: Summary of PSR flag bits on page 53 indicates the value of these bits for flag setting operations.

Table 6-2: Summary of PSR flag bits

Flag	Bit	Name	Description
N	31	Negative	Set to the same value as bit[31] of the result. For a 32-bit signed integer, bit[31] being set indicates that the value is negative.
Z	30	Zero	Set to 1 if the result is zero, otherwise it is set to 0.
C	29	Carry	Set to the carry-out value from result, or to the value of the last bit shifted out from a shift operation.
V	28	Overflow	Set to 1 if signed overflow or underflow occurred, otherwise it is set to 0.

The C flag is set if the result of an unsigned operation overflows the 32-bit result register. This bit might be used to implement 64-bit (or longer) arithmetic using 32-bit operations, for example.

The V flag operates in the same way as the C flag, but for signed operations. `0x7FFFFFFF` is the largest signed positive integer that can be represented in 32 bits. If, for example, you add 2 to this value, you will produce `0x80000001`, a large negative number. The V bit is set to indicate the overflow or underflow, from bit [30] to bit [31].

6.2 Data processing operations

These are essentially the fundamental arithmetic and logical operations of the processor. Multiplies can be considered a special case of these - they typically have slightly different format and rules and are executed in a dedicated unit of the processor.

The ARM processors can only perform data processing on registers, never directly on memory. Data processing instructions (for the most part) use one destination register and two source operands. The basic format can be considered to be the opcode, optionally followed by a condition code, optionally followed by S (set flags), as follows:

```
Operation{cond}{s} Rd, Rn, Operand2
```

Table 6-3: Summary of data processing operations in assembly language on page 54 summarizes the data processing assembly language instructions, giving their mnemonic opcode, operands and a brief description of their function.

Table 6-3: Summary of data processing operations in assembly language

Opcode	Operands	Description	Function
Arithmetic operations			
ADC	Rd, Rn, Op2	Add with carry	Rd = Rn + Op2 + C
ADD	Rd, Rn, Op2	Add	Rd = Rn + Op2
MOV	Rd, Op2	Move	Rd = Op2
MVN	Rd, Op2	Move NOT	Rd = ~Op2
RSB	Rd, Rn, Op2	Reverse Subtract	Rd = Op2 - Rn
RSC	Rd, Rn, Op2	Reverse Subtract with Carry	Rd = Op2 - Rn - !C
SBC	Rd, Rn, Op2	Subtract with carry	Rd = Rn - Op2 -!C
SUB	Rd, Rn, Op2	Subtract	Rd = Rn - Op2
Logical operations			
AND	Rd, Rn, Op2	AND	Rd = Rn & Op2
BIC	Rd, Rn, Op2	Bit Clear	Rd = Rn & ~ Op2
EOR	Rd, Rn, Op2	Exclusive OR	Rd = Rn ^ Op2
ORR	Rd, Rn, Op2	OR	Rd = Rn | Op2 (OR NOT) Rd = Rn | ~Op2
Flag setting instructions			
CMP	Rn, Op2	Compare	Rn - Op2
CMN	Rn, Op2	Compare Negative	Rn + Op2
TEQ	Rn, Op2	Test EQuivalence	Rn ^ Op2
TST	Rn, Op2	Test	Rn & Op2

The purpose and function of many of these instructions should be apparent to most programmers, but some require additional explanation.

In the arithmetic operations, notice that the move operations MOV and MVN require only one operand (and this is treated as an operand 2 for maximum flexibility, as we shall see). RSB does a reverse subtract - that is to say it subtracts the first operand from the second operand. This instruction is required because the first operand is inflexible - it can only be a register value. So to write R0 = 100 - R1, you must use `RSB R0,R1,#100`, as `SUB R0,#100,R1` is an illegal instruction. The operations ADC, RSC and SBC perform additions and subtractions with carry. This lets you synthesize arithmetic operations on values larger than 32 bits.

The logical operations are essentially the same as the corresponding C operators. Notice the use of ORR rather than OR (this is because the original ARM instruction set had three letter acronyms for all data-processing operations). The BIC instruction does an AND of a register with the inverted value of operand 2. If, for example, you want to clear bit [11] of register R0, you can do it with the instruction `BIC R0, R0, #0x800`.

The second operand `0x800` has only bit [11] set to one, with all other bits at zero. The `BIC` instruction inverts this operand, setting all bits except bit [11] to logical one. ANDing this value with the value in R0 has the effect of clearing bit [11] and this result is then written back into R0.

The compare and test instructions modify the CPSR and have no other effect.

6.2.1 Operand 2 and the barrel shifter

The first operand for all data processing operations must always be a register. The second operand is much more flexible and can be either an immediate value (#x), a register (Rm), or a register shifted by an immediate value or register `Rm, shift #x` or `Rm, shift Rs`. There are five shift operations: logical left shift (LSL), logical right-shift (`LSR`), arithmetic right-shift (`ASR`), rotate-right (`ROR`) and rotate-right extended (`RRX`).

A right shift creates empty positions at the top of the register. In that case, you must differentiate between a logical shift, that inserts 0 into the most significant bit(s) and an arithmetic shift, that fills vacant bits with the sign bit, from bit [31] of the register. So an `ASR` operation might be used on a signed value, with `LSR` used on an unsigned value. No such distinction is required on left-shifts, that always insert 0 to the least significant position.

So, unlike many assembly languages, ARM assembly language does not require explicit shift instructions. Instead, the `MOV` instruction can be used for shifts and rotates. `R0 = R1 >> 2` is done as `MOV R0, R1, LSR #2`. Equally, it is common to combine shifts with `ADD`, `SUB` or other instructions. For example, to multiply R0 by 5, you might write:

```
ADD R0, R0, R0, LSL #2
```

A left shift of n places is effectively a multiply by 2 to the power of n, so this effectively makes `R0 = R0 + (4 × R0)`. A right shift provides the corresponding divide operation, although `ASR` rounds negative values differently than would division in C.

Apart from multiply and divide, another common use for shifted operands is array index look-up. Consider the case where R1 points to the base element of an array of `int` (32-bit) values and R2 is the index that points to the nth element in that array. You can obtain the array value with a single load instruction that uses the calculation R1 + (R2 × 4) to get the appropriate address. The following provides examples of differing operand 2 types used in ARM instructions.

```
add     R0, R1, #1              @ R0 = R1 + 1
add     R0, R1, R2              @ R0 = R1 + R2
add     R0, R1, R2, LSL #4      @ R0 = R1 + R2<<#4
add     R0, R1, R2, LSL R3      @ R0 = R1 + R2<<R3
```

6.2.2 Multiplication operations

The multiply operations are readily understandable. A key limitation is that there is no scope to multiply by an immediate value. Multiplies operate only on values in registers. Multiplication by a constant might require that constant to be loaded into a register first. Later versions of the ARM processor add significantly more multiply instructions, giving a range of possibilities for 8, 16 and 32-bit data. We will consider these in Integer SIMD instructions when looking at the `DSP` instructions.

Table 6-4: Summary of multiplication operations in assembly language on page 56 summarizes the multiplication assembly language instructions, giving their mnemonic opcode, operands and a brief description of their function.

Table 6-4: Summary of multiplication operations in assembly language

Opcode	Operands	Description	Function
Multiplies			
`MLA`	Rd, Rn, Rm, Ra	Multiply accumulate (MAC)	`Rd = Ra + (Rn × Rm)`
`MLS`	Rd, Rn, Rm, Ra	Multiply and Subtract	`Rd = Ra - (Rm × Rn)`
`MUL`	Rd, Rn, Rm	Multiply	`Rd = Rn × Rm`
`SMLAL`	RdLo, RdHi, Rn, Rm	Signed 32-bit multiply with a 64-bit accumulate	`RdHiLo += Rn × Rm`
`SMULL`	RdLo, RdHi, Rn, Rm	Signed 32-bit multiply with 64-bit result	`RdHiLo = Rn × Rm`
`UMLAL`	RdLo, RdHi, Rn, Rm	Unsigned 32-bit MAC with a 64-bit result.	`RdHiLo += Rn × Rm`
`UMULL`	RdLo, RdHi, Rn, Rm	Unsigned 32-bit multiply with a 64-bit result	`RdHiLo = Rn × Rm`

6.2.3 Additional multiplies

We saw in the data-processing instructions that we have the ability to multiply one 32-bit register with another, to produce either a 32-bit result or a 64-bit signed or unsigned result.

In all cases, there is the option to accumulate a 32-bit or 64-bit value into the result. Additional multiply instructions have been added. There are signed most-significant word multiplies, `SMMUL`, `SMMLA` and `SMMLS`. These perform a 32 × 32-bit multiply in which the result is the top 32 bits of the product, with the bottom 32 bits discarded. The result can be rounded by applying an R suffix, otherwise it is truncated. The `UMAAL` (Unsigned Multiply Accumulate Accumulate Long) instruction performs a 32 × 32-bit multiply and adds in the contents of two 32-bit registers.

6.2.4 Hardware divide operations

The `SDIV` and `UDIV` hardware divide instructions are available in all implementations of the ARMv7-R architecture profile, but only in some ARMv7-A implementations.

`SDIV` performs a signed 32-bit integer division.

`UDIV` performs an unsigned 32-bit integer division.

6.3 Memory instructions

ARM processors perform Arithmetic Logic Unit (ALU) operations only on registers. The only supported memory operations are the load (that read data from memory into registers) or store (that write data from registers to memory). A `LDR` and `STR` can be conditionally executed, in the same fashion as other instructions.

You can specify the size of the Load or Store transfer by appending a B for Byte, H for Halfword, or D for doubleword (64 bits) to the instruction, for example, `LDRB`. For loads only, an extra S can be used to indicate a signed byte or halfword (SB for Signed Byte or SH for Signed Halfword).

This approach can be useful, because if you load an 8-bit or 16-bit quantity into a 32-bit register you must decide what to do with the most significant bits of the register. For an unsigned number, you zero-extend, that is, you write the most significant 16 or 24 bits of the register to zero. But for a signed number, it is necessary to copy the sign bit (bit [7] for a byte, or bit [15] for a halfword) into the top 16 (or 24) bits of the register.

6.3.1 Addressing modes

There are multiple addressing modes that can be used for loads and stores. The number in parentheses refers to the examples below:

- Register addressing- the address is in a register (1).
- Pre-indexed addressing - an offset to the base register is added before the memory access. The base form of this is `LDR Rd, [Rn, Op2]`. The offset can be positive or negative and can be an immediate value or another register with an optional shift applied.(2),(3).
- Pre-indexed with write-back - this is indicated with an exclamation mark (!) added after the instruction. After the memory access has occurred, this updates the base register by adding the offset value (4).
- Post-index with write-back - here, the offset value is written after the square bracket. The address from the base register only is used for the memory access, with the offset value added to the base register after the memory access (5).

```
(1)  LDR     R0, [R1]              @ address pointed to by R1

(2)  LDR     R0, [R1, R2]          @ address pointed to by R1 + R2

(3)  LDR     R0, [R1, R2, LSL #2]  @ address is R1 + (R2*4)

(4)  LDR     R0, [R1, #32]!        @ address pointed to by R1 + 32, then R1:=R1 +
 32

(5)  LDR     R0, [R1], #32    @ read R0 from address pointed to by R1, then R1:=R1 +
 32
```

6.3.2 Multiple transfers

Load and Store Multiple instructions enable successive words to be read from or written to memory. These are extremely useful for stack operations and for memory copying. Only word values can be transferred in this way and a word aligned address must be used.

The operands are a base register with a list of registers between braces. The optional ! denotes write-back of the base register. The register list is comma separated, with hyphens used to indicate ranges. The order in which the registers are loaded or stored has nothing to do with the order specified in this list. Instead, the operation proceeds in a fixed fashion, in increasing register order, with the lowest numbered register always mapped to the lowest address.

For example:

```
LDMIA   R10!, { R0-R3, R12 }
```

This instruction reads five registers from the addresses pointed to by register (R10) and because write-back is specified, increments R10 by 20 (5 × 4 bytes) at the end.

The instruction must also specify how to proceed from the base register Rd. The four possibilities are: IA/IB (Increment After/Before) and DA/DB (Decrement After/Before). These can also be specified using aliases (FD, FA, ED and EA) that work from a stack point of view and specify whether the stack pointer points to a full or empty top of the stack, and whether the stack ascends or descends in memory.

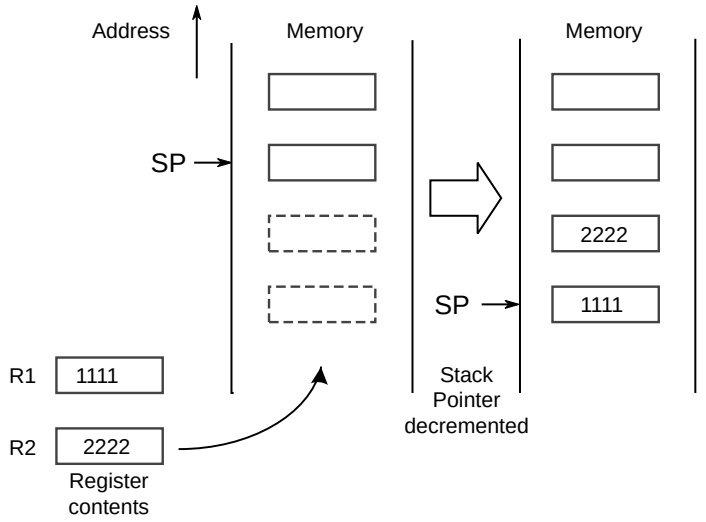
By convention, only the Full Descending (FD) option is used for stacks in ARM processor based systems. This means that the stack pointer points to the last filled location in stack memory and will decrement with each new item of data pushed to the stack.

For example:

```
STMFD    sp!, {r0-r5}  ; Push onto a Full Descending Stack
LDMFD    sp!, {r0-r5}  ; Pop from a Full Descending Stack
```

Figure 6-1: Stack push operation on page 59 shows a push of two registers to the stack. Before the STMFD (PUSH) instruction is executed, the stack pointer points to the last occupied word of the stack. After the instruction is completed, the stack pointer has been decremented by 8 (two words) and the contents of the two registers have been written to memory, with the lowest numbered register being written to the lowest memory address.

Figure 6-1: Stack push operation



6.4 Branches

The instruction set provides a number of different kinds of branch instruction. For simple relative branches (those to an offset from the current address), the `B` instruction is used. Calls to subroutines, where it is necessary for the return address to be stored in the link register, use the `BL` instruction.

If you want to change instruction set (from ARM to Thumb or Thumb to ARM), use `BX`, or `BLX`.

You can also specify the PC as the destination register for the result of normal data processing operations such as `ADD` or `SUB`, but this is generally deprecated and is unsupported in Thumb. An additional type of branch instruction can be implemented using either a load (`LDR`) with the PC as the target, load multiple (`LDM`), or stack-pop (`POP`) instruction with PC in the list of registers to be loaded.

Thumb has the compare and branch instruction. This fuses a `CMP` instruction and a conditional branch, but does not change the CPSR condition code flags. There are two opcodes, `CBZ` (compare and branch to label if Rn is zero) and `CBNZ` (compare and branch to label if Rn is not zero). These instructions can only branch forward between 4 and 130 bytes. Thumb also has the `TBB` (Table Branch Byte) and `TBH` (Table Branch Halfword) instructions. These instructions read a value from a table of offsets (either byte or halfword size) and perform a forward PC-relative branch of twice the value of the byte or the halfword returned from the table. These instructions require the base address of the table to be specified in one register, and the index in another.

Knowledge of the processor behavior with branches can be useful when writing highly optimized code. The hardware performance monitor counters can generate information about the numbers of branches correctly or incorrectly predicted.

When moving or modifying code at an address from which code has already been executed in the system, it might be necessary (and is always prudent) to remove stale entries from the branch history logic by using the CP15 instruction that invalidates all entries.

6.4.1 Direct and indirect branches

Branches can be split into two categories, direct and indirect branches. Direct branches are PC relative, and branch to an offset from the current address. The range of a direct branch is limited, for example, +/-32MB for an ARM `B` or `BL` instruction. Because the branch destination is PC-relative, it can be determined exactly at an early stage of the pipeline.

```
B <label>
BL <label>
BLX <label>
TBB [Rn, Rm]
TBH [Rn, Rm]
```

Indirect branches perform an absolute branch, so can branch to any location in the address space. However, because the destination is specified in a register or loaded from memory, the destination cannot be easily predicted by the processor.

```
BX <Rd>
LDR pc, [Rd]
ADD pc, Rn, Rm
```

6.5 Branch prediction

Many branch instructions are conditional. For conditional branches, whether the branch should be taken cannot be determined until the instruction is executed. The processor makes a prediction about whether the branch will be taken and fetches based on the prediction. The processor must also be able to detect when it gets the prediction wrong, and re-fetch from the correct location.

Branch prediction logic is an important factor in achieving high throughput in Cortex-R series processors. With no branch prediction, you would have to wait until a conditional branch executes before you could determine where to fetch the next instruction from.

The branch prediction logic predicts:

- Whether there is a branch instruction at a given address.
- The type of the branch:
  - Unconditional or conditional.
  - Immediate or load.
  - Normal branch, function call, or function return.
- The target address and the state of the branch, either ARM or Thumb.
- The direction of conditional branch, either taken or not taken.

There are two branch prediction methods:

- Static branch prediction.
- Dynamic branch prediction.

6.5.1 Static branch prediction

The static branch prediction method is simple as it requires no prior information about the branch. The prediction happens at the decode stage. A fetch decision cannot be made before this stage. The first time that a conditional branch instruction is fetched, there is little information on which to base a prediction about the address of the next instruction. It speculatively takes backward branches rather than forward branches.

A backward branch has a target address that is lower than its own address. It can therefore look at a single opcode bit to determine the branch direction. This technique can give reasonable prediction accuracy because of the prevalence of loops, where backward-pointing branches are taken more often than not taken.

6.5.2 Dynamic branch prediction

Because of the longer pipeline length, complex branch prediction schemes, such as dynamic prediction, gives better prediction accuracy. Dynamic prediction hardware can reduce the average branch penalty by making use of historical information about whether conditional branches were taken or not taken on previous execution. It can speculatively fetch a chosen branch of the execution code.

A Branch Target Address Cache (BTAC), in the Cortex-R7 processor, is a cache that holds information about previous branch instruction execution. Dynamic branch prediction avoids unnecessary instruction cache lookup and memory accesses. The prediction quality is higher for previously seen branches. By default, the processor uses dynamic branch prediction. If there is no history information, then it uses static branch prediction.

The processor must still evaluate the condition code attached to a branch instruction. If the branch prediction hardware predicts correctly, the pipeline does not have to be stalled. If the branch prediction hardware speculation was wrong, the processor will flush the pipeline and refill it.

6.5.3 Return stack prediction

For most branch instructions, the target address is fixed and encoded in the instruction. However, there is a class of branches where the branch target destination cannot be determined by looking at the instruction. For example, if you perform a data processing operation that modifies the PC (for example, `MOV`, `ADD` or `SUB`) you must wait for the ALU to evaluate the result before you can know the

branch target. Similarly if you load the PC from memory, using an `LDR`, `LDM` or `POP` instruction, you cannot know the target address until the load completes.

Such branches are called indirect branches and generally cannot be predicted in hardware. A common indirect branch case is the function return, though this can be optimized, using a Last In First Out (LIFO) stack in the pre-fetch hardware, the return stack.

The Return Stack for the Cortex-R4 and Cortex-R5 processors consists of a four entry LIFO buffer. The Cortex-R7 processor has a eight stack FIFO buffer. When a function call instruction is executed, the generated LR is pushed onto the LIFO buffer. When a function return is detected, the pipeline predicts that the destination is the top entry of the LIFO buffer.

Recognized function calls:

```
BL immediate
BLX immediate
BLX Rm
```

Recognized function returns:

```
POP   {..,pc}
LDMIB Rn{!}, {..,pc}
LDMDA Rn{!}, {..,pc}
LDMDB Rn{!}, {..,pc}
LDR   pc, [sp], #4
BX    Rm
BX    LR
```

Not all possible return sequences are predicted. For example, the return stack does not recognize `MOV pc, lr` that might be present in legacy code.

When a function call (`BL` or `BLX`) instruction is executed, you should enter the address of the following instruction into this stack. When you encounter an instruction that is recognized as a function return instruction, you can speculatively pop an entry from the stack and start fetching instructions from that address. When the return instruction actually executes, the hardware compares the address generated by the instruction with that predicted by the stack. If there is a mismatch, the pipeline is flushed and you restart from the correct location.

The return stack is of a fixed size. If a particular code sequence contains a large number of nested function calls, an eight entry return stack can predict only the first eight function returns.

6.6 Integer SIMD instructions

This section describes the SIMD (Single Instruction, Multiple Data) operations added in the ARMv6 architecture. SIMD is one of four classifications of computer architectures defined by Michael J. Flynn in 1966 based on the number of instruction and data streams available in the architecture.

These instructions provide the ability to pack, extract and unpack 8-bit and 16-bit quantities within 32-bit registers and to perform multiple arithmetic operations such as add, subtract, compare or multiply to such packed data, with a single instruction.

Note

The SIMD instructions are part of the ARM and Thumb instruction set.

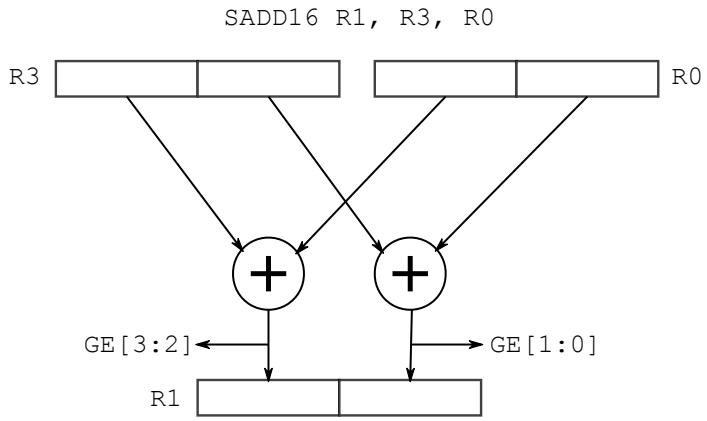
6.6.1 Integer register SIMD instructions

SIMD operations make use of the GE (greater than or equal) flags within the CPSR. These are distinct from the normal condition flags. There is a flag corresponding to each of the four byte positions within a word. Normal data processing operations produce one value and set the N, Z, C and V flags (as seen in Program Status Registers). The SIMD operations produce up to four outputs and set only the GE flags, to indicate overflow. The `MSR` and `MRS` instructions can be used to write or read these flags directly.

The general form of the SIMD instructions are that subword quantities in each register are operated on in parallel (for example, four `ADDS` on four bytes can be performed) and the GE flags are set or cleared according to the results of the instruction. Different types of add and subtract can be specified using appropriate prefixes. For example, `QADD16` performs saturating addition on halfwords within a register. `SADD/UADD8` and `SSUB/USUB8` set the GE bits individually while `SADD/UADD16` and `SSUB/USUB16` set GE bits [3:2] together based on the top halfword result, and [1:0] together on the bottom halfword result.

Also available are the `ASX` and `SAX` class of instructions, that reverse halfwords of one operand and add/subtract or subtract/add parallel pairs. Like the previously described `ADD` and Subtract instructions, these exist as unsigned (`UASX/USAX`), signed (`SASX/SSAX`) and saturated (`QASX/QSAX`) versions.

Figure 6-2: ADD v6 SIMD example



The SADD16 instruction shown in Figure 6-2: ADD v6 SIMD example on page 64 shows how two separate addition operations are performed by a single instruction. The top halfwords of registers R3 and R0 are added, with the result going into the top halfword of register R1 and the bottom halfwords of registers R3 and R0 are added, with the result going into the bottom halfword of register R1. GE[3:2] bits in the CPSR are set based on the top halfword result and GE[1:0] based on the bottom halfword result. In each case the carry flag is duplicated in the specified pair of bits. The two operations are entirely separate. In particular, there is no overflow from bit 15 (the top of the lower addition) to bit 16 (the bottom of the higher).

6.6.2 Integer register SIMD multiplies

Like the other SIMD operations, these operate in parallel, on subword quantities within registers. The instruction can also include an accumulate option, with add or subtract being able to be specified. The instructions are SMUAD (SIMD multiply and add with no accumulate), SMUSD (SIMD multiply and subtract with no accumulate), SMLAD (multiply and add with accumulate) and SMLSD (multiply and subtract with accumulate).

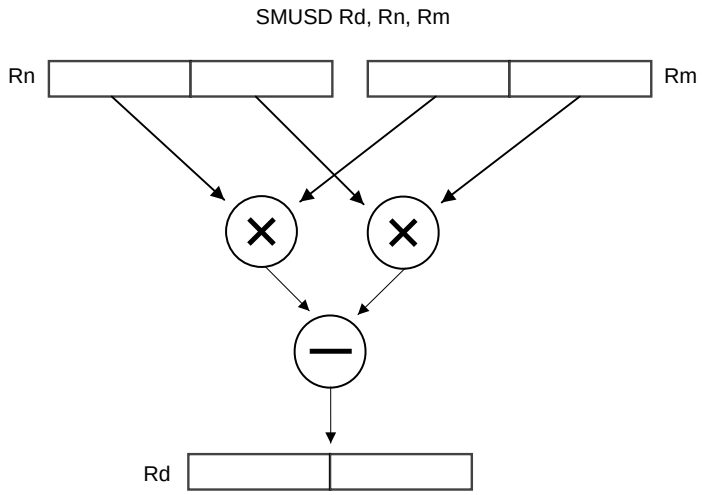
Adding an L (long) before D indicates 64-bit accumulation.

Using the X (eXchange) suffix indicates halfwords in Rm are swapped before calculation.

The Q flag is set if accumulation overflows.

The SMUSD instruction shown in Figure 6-3: v6 SIMD signed dual multiply subtract example on page 65 performs two signed 16-bit multiplies (top × top and bottom × bottom) and then subtracts the two results. This kind of operation is useful when performing operations on complex numbers (with a real and imaginary component), a common task for filter algorithms.

Figure 6-3: v6 SIMD signed dual multiply subtract example

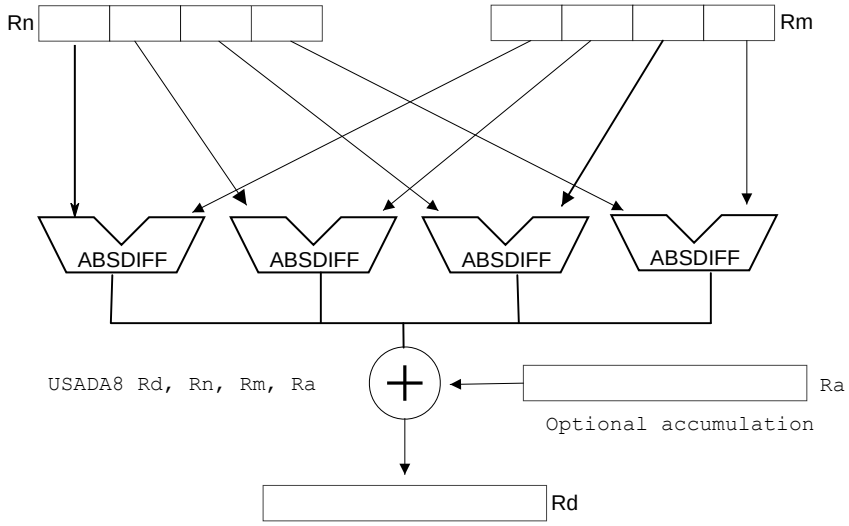


6.6.3 Sum of absolute differences

Calculating the sum of absolute differences is a key operation in the motion vector estimation component of common video codecs and is carried out over arrays of pixel data.

The `USADA8 Rd, Rn, Rm, Ra` instruction is illustrated in Figure 6-4: Sum of absolute differences on page 65. It calculates the sum of absolute differences of the bytes within a word in registers `Rn` and `Rm`, adds in the value stored in `Ra` and places the result in `Rd`.

Figure 6-4: Sum of absolute differences

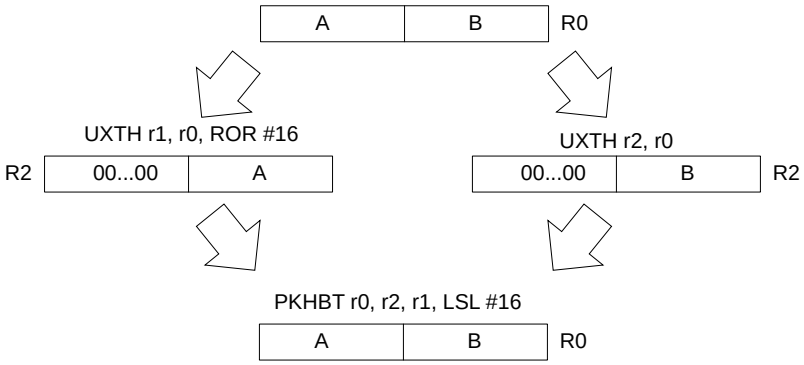


6.6.4 Data packing and unpacking

Packed data is common in many video and audio codecs (video data is usually expressed as packed arrays of 8-bit pixel data, audio data can use packed 16-bit samples), and also in network protocols. Before additional instructions were added in the ARMv6 architecture, this data had to be either loaded with `LDRH` and `LDRB` instructions or loaded as words and then unpacked using Shift and Bit Clear operations; both are relatively inefficient. Pack (`PKHBT`, `PKHTB`) instructions enable 16-bit or 8-bit values to be extracted from any position in a register and packed into another register. Unpack instructions (`UXTH`, `UXTB`, plus many variants, including signed, with addition) can extract 8-bit or 16-bit values from any bit position within a register.

This enables sequences of packed data in memory to be loaded efficiently using word or doubleword loads, unpacked into separate register values, operated on and then packed back into registers for efficient writing out to memory.

Figure 6-5: Packing and unpacking of 16-bit data in 32-bit registers



In the simple example shown in Figure 6-5: Packing and unpacking of 16-bit data in 32-bit registers on page 66, R0 contains two separate 16-bit values, denoted A and B. You can use the `UXTH` instruction to unpack the two halfwords into registers for additional processing and you can then use the `PKHBT` instruction to pack halfword data from two registers into one. It would be possible to replace the unpack instruction in each case with a `MOV` and either `LSL` or `LSR` instructions, but in this case you use a single instruction intended to work on parts of registers.

6.6.5 Byte selection

The `SEL` instruction enables us to select each byte of the result from the corresponding byte in either the first or the second operand, based on the value of the GE[3:0] bits in the CPSR. The packed data arithmetic operations set these bits as a result of add or subtract operations, and `SEL` can be used after these to extract parts of the data - for example, to find the smaller of the two bytes in each position.

6.7 Saturating arithmetic

Saturated arithmetic is commonly used in audio and video codecs. Calculations that return a value higher (or lower) than the largest positive (or negative) number that can be represented do not overflow. Instead the result is set to the largest positive or negative value (saturated). The ARM instruction set includes a number of instructions that enables easy implementation of such algorithms.

6.7.1 Saturated math instructions

The ARM saturated arithmetic instructions can operate on byte, word or halfword sized values. For example, the 8 of the `QADD8` and `QSUB8` instructions indicate that they operate on byte sized values. The result of the operation is saturated to the largest possible positive or negative number. If the result would have overflowed and has been saturated, the overflow flag (CPSR Q bit) is set. This flag is said to be sticky. When set it will remain set until explicitly cleared by a write to the CPSR.

The instruction set provides special instructions with this behavior, `QSUB` and `QADD`. Additionally, `QDSUB` and `QDADD` are provided in support of Q15 or Q31 fixed point arithmetic. These instructions double and saturate their second operand before performing the specified add or subtract.

The Count Leading Zeros (`CLZ`) instruction returns the number of 0 bits before the most significant bit that is set. This can be useful for normalization and for certain division algorithms. To saturate a value to a specific bit position (effectively saturate to a power of two), you can use the `USAT` or `SSAT` (unsigned or signed) saturate operations. `USAT16` and `SSAT16` permit saturation of two halfword values packed within a register.

6.8 Miscellaneous instructions

The remaining instructions cover coprocessor, supervisor call, PSR modification, byte reversal, cache preload, bit manipulation and a few others.

6.8.1 Coprocessor instructions

Coprocessor instructions occupy part of the ARM instruction set. Up to 16 coprocessors can be implemented, numbered 0 to 15 (CP0, CP1 ... CP15). These can either be internal (built-in to the processor) or connected externally, through a dedicated interface. Use of external coprocessors is uncommon in older processors and is not supported at all in the Cortex-R series.

- Coprocessor 15 is a built-in coprocessor that provides control over many processor features, including cache and MPU.
- Coprocessor 14 is a built-in coprocessor that controls the hardware debug facilities of the processor, such as breakpoint units (described in Debug).
- Coprocessors 10 and 11 give access to the floating-point hardware in the system (described in Floating-Point).

If a coprocessor instruction is executed, but the appropriate coprocessor is not present in the system, an undefined instruction exception occurs.

There are five classes of coprocessor instruction

- `CDP` - initiate a coprocessor data processing operation.
- `MRC` - move to ARM register from coprocessor register.
- `MCR` - move to coprocessor register from ARM register.
- `LDC` - load coprocessor register from memory.
- `STC` - store from coprocessor register to memory.

Multiple register and other variants of these instructions are also available:

- `MRRC` - transfers a value from a Coprocessor to a pair of ARM registers.
- `MCCR` - transfers a pair of ARM register to a coprocessor.
- `LDCL` - reads multiple words of memory from a coprocessor register,
- `STCL` - writes multiple words of memory to a coprocessor register,

6.8.2 SVC

The `SVC` (supervisor call) instruction, when executed, causes a supervisor call exception.

This is described in Exceptions and Interrupts. The instruction includes a 24-bit (ARM) or 8-bit (Thumb) number value, that can be examined by the `SVC` handler code. Through the `SVC` mechanism, an operating system can specify a set of privileged operations that applications running in User mode can request. This instruction was originally called `SWI` (Software Interrupt).

6.8.3 PSR modification

Several instructions enable the PSR to be written to, or read from:

- `MRS` transfers the CPSR or SPSR value to a general purpose register.

  `MSR` transfers a general purpose register to the CPSR or SPSR.

  Either the whole status register, or part of it can be updated. In User mode, all bits can be read, but only the condition flags (`_f`) are permitted to be modified.
- In a privileged mode the Change Processor State (`CPS`) instruction can be used to directly modify the mode and interrupt-enable or disable (I and F) bits in the CPSR.
- `SETEND` modifies a single CPSR bit, the E (Endian) bit. This can be used in systems with mixed endian data to temporarily switch between little- and big-endian data access.

6.8.4 Bit manipulation

There are instructions that permit bit manipulation of values in registers:

- The Bit Field Insert (`BFI`) instruction enables a series of adjacent bits from the bottom of one register (specified by supplying a width value and LSB position) to be placed into any position in the destination register.
- The Bit Field Clear (`BFC`) instruction enables adjacent bits within a register to be cleared.
- The `SBFX` and `UBFX` instructions (Signed and Unsigned Bit Field Extract) copy adjacent bits from one register to the least significant bits of a second register, and sign extend or zero extend the value to 32 bits.
- The `RBIT` instruction reverses the order of all bits within a register.

6.8.5 Cache preload

Two instructions are provided, `PLD` (data cache preload) and `PLI` (instruction cache preload). Both instructions act as hints to the memory system that an access to the specified address is likely to occur soon. An illegal address specified as a parameter to the `PLD` instruction will not result in a data abort exception.

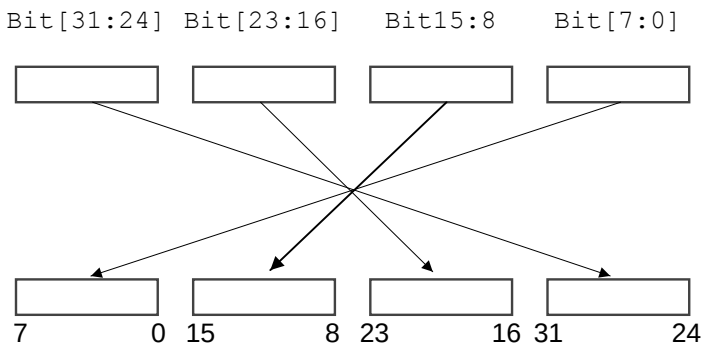
6.8.6 Byte reversal

Instructions to reverse byte order can be useful for dealing with quantities of the opposite endianness or other data re-ordering operations.

- The `REV` instruction reverses the bytes in a word
- The `REV16` reverses the bytes in each halfword of a register
- The `REVSH` reverses the bottom two bytes, and sign extends the result to 32 bits.

Figure 6-6: Operation of the REV instruction on page 69 illustrates the operation of the `REV` instruction, showing how four bytes within a register have their ordering within a word reversed.

Figure 6-6: Operation of the REV instruction



6.8.7 Other instructions

A few other instructions are available:

- The breakpoint instruction (`BKPT`) will either cause a prefetch abort (see Types of exception) or cause the processor to enter debug state (depending on the whether the processor is configured for monitor or halt mode debug). This instruction is used by debuggers. See Debug.
- Wait For Interrupt (`WFI`) puts the processor into standby mode, described in Power Management. The processor stops execution until woken by an interrupt or debug event. If `WFI` is executed with interrupts disabled, an interrupt will still wake the processor, but no interrupt exception is taken. The processor proceeds to the instruction after the `WFI`. In older ARM processors, `WFI` was implemented as a CP15 operation.
- A **`NOP`** instruction (no-operation) does nothing. It may or may not take time to execute, so the **`NOP`** instruction should not be used to insert timing delays into code. It is intended to be used as padding.
- A Wait for Event (`WFE`) instruction puts the core into standby mode in a similar way to `WFI`. The core will sleep until woken by an event generated by another core executing a `SEV` instruction. An interrupt or a bug event will also cause the core to wake up.
- The `SEV` (Send Event) instruction is used to generate wake-up events that might wake-up other cores in the cluster.

7. Floating-Point

All computer programs deal with numbers. Floating-point numbers, however, can sometimes appear counter-intuitive to programmers who are not familiar with their detailed implementation. Before looking at floating-point implementation on ARM processors, a short overview of floating-point fundamentals is included. Programmers with prior floating-point experience might want to skip the following section.

7.1 Floating-point basics and the IEEE-754 standard

The IEEE-754 standard is the reference for almost all modern computer floating-point mathematics implementations, including ARM floating-point systems. The original IEEE-754-1985 standard was updated with the publication of IEEE-754-2008. The standard defines precisely what result is produced by each of the fundamental floating-point operations over all of the possible input values. It describes what a compliant implementation should do with respect to rounding of results that cannot be expressed precisely. A simple example of such a calculation would be 1.0 ÷ 3.0, that would require an infinite number of digits to express precisely in decimal or binary notation.

IEEE-754 provides a number of different rounding options to cope with this (round towards positive infinity, round towards negative infinity, round toward zero, and two forms of round to nearest, see Rounding algorithms). IEEE-754 also specifies the outcome when an exceptional operation occurs. This means a calculation which potentially represents a problem. These conditions can be tested, either by querying the FPSCR (on ARM processors) or by setting up trap handlers (on some systems). Examples of exceptional operations are as follows:

Overflow
: A result that is too large to represent.

Underflow
: A result that is so small that precision is lost.

Inexact
: A result that cannot be represented without some loss of precision. It is clear that many floating-point calculations will fall into this category.

Invalid
: For example, attempting to calculate the square root of a negative number.

Division by zero
: Attempting to divide by zero.

The specification also describes what action should be taken when one of the above exceptional operations is detected. Possible outcomes include the generation of a NaN (Not a Number) result for invalid operations, positive or negative infinity, for overflow or division by zero, or denormalized numbers in the case of underflow. The standard defines what results should be produced if subsequent floating-point calculations operate on NaN or infinities.

One of the things that IEEE-754 defines is how floating-point numbers are represented within the hardware. Floating-point numbers are typically represented using either single precision (32-bit) or double-precision (64-bit). VFP supports single-precision (32-bit) and double-precision (64-bit) formats in hardware. In addition, VFPv3 can have half-precision extensions to enable 16-bit values to be used for storage.

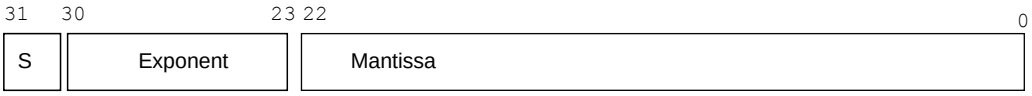
Floating-point formats use the available space to store three pieces of information about a floating-point number:

- A sign bit (S) that shows whether the number is positive (0) or negative (1).
- An exponent giving its order of magnitude.
- A mantissa giving the fractional binary digits of the number.

For a single precision `float`, for example, bit [31] of the word is the sign bit [S], bits [30: 23] give the exponent and bits [22:0] give the mantissa. See Figure 7-1: Single precision floating-point format on page 72.

The value of the number is then $\pm m \times 2^{exp}$, where m is derived from the mantissa and exp is derived from the exponent.

Figure 7-1: Single precision floating-point format



The mantissa is not generated by directly taking the 23-bit binary value, but rather, it is interpreted as being to the right of the binary point, with a 1 present to the left. In other words, the binary mantissa must be greater than or equal to one and less than two. In the case where the number is zero, this is represented by setting all of the exponent and mantissa bits to 0. There are other special-case representations, for positive and negative infinity, and for the not-a-number (NaN) values. A special case is that of denormalized values.

The sign bit lets us distinguish positive and negative infinity and NaN representations. Similarly, the 8-bit exponent value is used to give a value in the range +128 to -127, so there is an implicit offset of -127 in the encoding. Table 7-1: Single precision floating-point representation on page 72 summarizes this.

Table 7-1: Single precision floating-point representation

Exponent	Mantissa	Description
-127	0	±0
-127	!=0	Subnormal values
128	0	±INFINITY
128	!=0	NaN values
Other	Any Normal values	+/-1.<mantissa> × 2^<exp>

Let's consider an example:

The decimal value +0.5 is represented as a single precision float by the hexadecimal value `0x3F000000`. This has a sign value of 0 (positive).

The value of the mantissa is 1.0, though the integral part (1) is implicit and is not stored. The exponent value is specified in bits [30:23] - that hold `0b01111110`, or 126 - offset by 127 to represent an exponent of -1.

The value is therefore given by $(-1)^{sign} \times mantissa \times 2^{exponent} = 1 \times 1 \times 2^{-1} = 0.5$ (decimal)

Denormal numbers are a special case. If you set the exponent bits to zero, you can represent very small numbers other than zero, by setting mantissa bits. Because normal values have an implied leading 1, the closest value to zero you can represent as a normal value is $\pm 2^{-126}$.

To get smaller numbers, the 1.m interpretation of the mantissa value is replaced with a 0.m interpretation. Now, the number's magnitude is determined only by bit positions. When using these extremely-small numbers, the available precision does not scale with the magnitude of the value. Without the implied 1 attached to the mantissa, all bits to the left of the lowest set bit are leading zeros, so the smallest representable number is 1.401298464e-45, represented by `0x00000001`.

For performance reasons, such denormal values are often ignored and are flushed to zero. This is strictly a violation of IEEE-754, but denormal values are used rarely enough in real programs that the performance benefit is worth more than correct handling of these extremely small numbers. Cortex processors with VFP enable code to select between flush-to-zero mode and full denormal support.

Because a 32-bit floating-point number has a 23-bit mantissa there are many values of a 32-bit `int` that if converted to 32-bit `float` cannot be represented exactly. This is referred to as loss of precision. If you convert one of these values to `float` and back to `int` you will get a different, nearby value. In the case of double-precision floating-point numbers, the exponent field has 11 bits (giving an exponent range from -1022 to +1023) and a mantissa field with 52 bits.

7.1.1 Rounding algorithms

The IEEE 754-1985 standard defines four different ways in which results can be rounded, as follows:

- Round to nearest (ties to even). This mode causes rounding to the nearest value. If a number is exactly midway between two possible values, it is rounded to the nearest value with a zero least significant bit.
- Round toward 0. This causes numbers to always be rounded towards zero (this can be also be viewed as truncation).
- Round toward +∞ .This selects rounding towards positive infinity.
- Round toward -∞. This selects rounding towards negative infinity.

The IEEE 754-2008 standard adds an additional rounding mode. In the case of round to nearest, it is now also possible to round numbers that are exactly halfway between two values, away from zero (in other words, upwards for positive numbers and downwards for negative numbers). This is

in addition to the option to round to the nearest value with a zero least significant bit. At present VFP does not support this rounding mode.

7.1.2 ARM VFP

VFP is an optional (but rarely omitted) extension to the instruction sets in the ARMv7-R architecture conforming to the IEEE 754 standard. It can be implemented with either thirty-two, or sixteen double-word registers. The terms VFPv3-D32 and VFPv3-D16 are used to distinguish between these two options. VFPv3 can also be optionally extended by the half-precision extensions that provide conversion functions in both directions between half-precision floating-point (16-bit) and single-precision floating-point (32-bit). These operations only permit half-precision floats to be converted to and from other formats.

VFPv4 adds both the half-precision extensions and the Fused Multiply-Add instructions to the features of VFPv3. In a Fused Multiply-Add operation, only a single rounding occurs at the end. This is one of the new facets of the IEEE 754-2008 specification. Fused operations can improve the accuracy of calculations that repeatedly accumulate products, such as matrix multiplication or dot product calculation. The VFP version supported by individual Cortex-R series processors is given in ARM Architecture and Processors.

There are a number of other VFP registers. These are listed below.

Floating-Point System ID Register (FPSID)

This can be read by system software to determine which floating-point features are supported in hardware.

Floating-Point Status and Control register (FPSCR)

This holds comparison results and flags for exceptions. Control bits select rounding options and enable floating-point exception trapping.

Floating-Point Exception Register (FPEXC)

The FPEXC register contains bits that enable system software that handles exceptions to determine what has happened.

Media and VFP Feature registers 0 and 1 (MVFR0 and MVFR1)

These registers enable system software to determine which Advanced SIMD and floating-point features are provided on the processor implementation.

User mode code can only access the FPCSR. One implication of this is that applications cannot read the FPSID to determine which features are supported unless the host OS provides this information. Linux provides this using `/proc/cpuinfo`, for example, but the information is not nearly as detailed as that provided by the VFP hardware registers.

Unlike ARM integer instructions, no VFP operations will affect the flags in the APSR directly. The flags are stored in the FPSCR. Before the result of a floating-point comparison can be used by the integer processor, the flags set by a floating-point comparison must be transferred to the APSR, using the `VMRS` instruction. This includes use of the flags for conditional execution, even of other VFP instructions. The following example shows a simple piece of code to illustrate this. The `VCMP` instruction performs a comparison the values in VFP registers d0 and d1 and sets FPSCR flags

as a result. These flags must then be transferred to the integer processor APSR, using the VMRS instruction. You can then conditionally execute instructions based on this.

```
VCMP d0, d1
VMRS APSR_nzcv, FPSCR
BNE  label
```

Flag meanings

The integer comparison flags support comparisons that are not applicable to floating-point numbers. For example, floating-point values are always signed, so there is no requirement for unsigned comparisons. On the other hand, floating-point comparisons can result in the unordered result (meaning that one or both operands was NaN, or Not a Number). IEEE-754 defines four testable relationships between two floating-point values, that map onto the ARM condition codes as follows:

Table 7-2: ARM APSR flags

IEEE-754 relationship	N	Z	C	V
Equal	0	1	1	0
Less Than (LT)	1	0	0	0
Greater Than (GT)	0	0	1	0
Unordered (At least one argument was NaN)	0	0	1	1

Compare with zero

Unlike the integer instructions, most VFP instructions can operate only on registers, and cannot accept immediate values encoded in the instruction stream. The VCMP instruction is a notable exception in that it has a special-case variant that enables quick and easy comparison with zero.

Interpreting the flags

When the flags are in the APSR, they can be used almost as if an integer comparison had set the flags. However, floating-point comparisons support different relationships, so the integer condition codes do not always make sense. Table 7-3: Interpreting the flags on page 75 describes floating-point comparisons rather than integer comparisons:

Table 7-3: Interpreting the flags

Code	Meaning (when set by cmp)	Flags tested
EQ	Equal to	Z = 1
NE	Not equal to.	Z = 0
CS	Carry set (identical to HS)	C = 1
HS	Unsigned higher or same	C = 1
CC	Carry clear (identical to LO)	C = 0
LO	Unsigned lower (identical to CC)	(C = 0) && (Z = 0)
MI	Negative.	N = 1
PL	Positive or zero.	N = 0

Code	Meaning (when set by cmp)	Flags tested
VS	Signed overflow.	V = 1
VC	No signed overflow.	V = 0
HI	Greater than (unsigned).	(C = 1) && (Z = 0)
LS	Less than or equal to (unsigned).	(C = 0) || (Z = 1)
GE	Greater than or equal to (signed).	N==V
LT	Less than (signed).	N!=V
GT	Greater than (signed).	(Z==0) && (N==V)
LE	Less than or equal to (signed).	(Z==1) || (N!=V)
AL (or omitted)	Always executed.	None tested.

It should be obvious that the condition code is attached to the instruction reading the flags, and the source of the flags makes no difference to the flags that are tested. Similarly, it is clear that the opposite conditions still hold. (For example, HS is still the opposite of LO.)

When set by CMP the flags generally have analogous meanings to the flags set by VCMP. For example, GT still means greater than. However, the unordered condition and the removal of the signed conditions can confuse matters. Often, for example, it is desirable to use LO, normally an unsigned less than check, in place of LT, because it does not match in the unordered case.

7.1.3 Instructions

VFP instructions are provided that perform arithmetic and data processing, load and stores to memory, and register transfers (between VFP registers and to or from ARM registers).

These instructions are encoded with ARM coprocessor instructions, but are typically viewed as part of the main instruction set, rather than as coprocessor operations. VFP offers all the common arithmetic operations, format conversions, a few complex arithmetic operations (for example, Multiply accumulate, VMLA, and square root, VSQRT), along with memory access instructions.

7.1.4 VFP support in GCC

Use of VFP is fully supported by GCC (although some builds can be configured to default to assume no VFP support, in which case floating-point calculations will use library code).

The main option to use for VFP support is:

- `-mfpu=vfp` specifies that the target has VFP hardware.

Other options can be used to specify support for a specific VFP implementation on an ARM Cortex-R series processor:

- `-mfpu=vfpv3`
- `-mfpu=vfpv3-d16`

These options can be used for code that will run only on these VFP implementations, and do not require backward compatibility with older VFP implementations.

The options that specify which ABI to use to enable the use of VFP:

- `-mfloat-abi=softfp`
- `-mfloat-abi=hard`

`softfp` uses a Procedure Call Standard compatible with software floating-point, and so provides binary compatibility with legacy code. This permits running older `soft float` code with new libraries that support hardware floating-point, but still makes use of hardware floating-point registers between function calls. `hard` has floating-point values passed in floating-point registers. This is more efficient but is not backward compatible with the `softfp` ABI variant. Particular care is required with libraries, including the C platform library.

C programmers should note that there can be a significant function call overhead when using `-mfloat-abi=softfp`, if many floating-point values are being passed.

7.1.5 Enabling VFP

If an ARMv7 processor includes VFP hardware, it must be explicitly enabled before applications can make use of it.

- The EN bit in the FPEXC register must be set.
- Access to CP10 and CP11 must be enabled in the Coprocessor Access Control Register (CP15.CACR). This can be done on demand by the operating system.

7.1.6 VFP in the Cortex-R processors

The Cortex-R4F, Cortex-R5F, and Cortex-R7 processors implement the VFPv3-D16 floating-point architecture and the Common VFP Sub-Architecture v2. They are IEEE-754 standard compliant. In the Cortex-R7 processors, each core has the option to implement the Floating-Point Unit (FPU).

- The Cortex-R4F processor implements a floating-point unit, with support for single-precision and double-precision floats.
- The Cortex-R5F processor has the option to implement the full FPU, with support for single-precision and double-precision floats or an optimized single-precision only FPU.
- The Cortex-R7 processor has the option to implement the full FPU, with support for single-precision, half-precision and double-precision floats or an optimized single-precision and half-precision only FPU.

7.2 VFP support in the ARM Compiler

Use of VFP is fully supported by the ARM Compiler (although some builds can be configured by default to assume no VFP support, in which case floating-point calculations will use library code).

The main option to use with the ARM Compiler for VFP support is:

- `--fpu=name` that lets you specify the target floating-point hardware.

The options used to specify support for a specific VFP implementation on an ARM Cortex-R series processor are:

- `--fpu=vfpv3`
- `--fpu=vfpv3_d16`

These options can be used for code that will run only on these VFP implementations, and do not require backward compatibility with older VFP implementations. Use `--fpu=list` to see the full list of FPUs supported.

The following options can be used for linkage support:

- `--apcs=/hardfp` generates code for hardware floating-point linkage
- `--apcs=/softfp` generates code for software floating-point linkage

Hardware floating-point linkage uses the FPU registers to pass the arguments and return values. Software floating-point linkage means that the parameters and return value for a function are passed using the ARM integer registers R0 to R3 and the stack. `--apcs=/hardfp` and `--apcs=/softfp` interact with or override explicit or implicit use of `--fpu`.

To compile with or without hard floating point, ARM Compiler 5 provides these compile switches:

- `--cpu=Cortex-R4` (no hardfp)
- `--cpu=Cortex-R4F` (hardfp)
- `--cpu=Cortex-R5` (no hardfp)
- `--cpu=Cortex-R5F` (hardfp)
- `--cpu=Cortex-R7` (hardfp)
- `--cpu=Cortex-R7.no_vfp` (no hardfp)

7.3 Floating-point optimization

This section contains some suggestions for developers writing FP assembly code. Some caution is required when applying these points, as recommendations can be specific to a particular piece

of hardware. A code sequence that is optimal for one processor can be sub-optimal on different hardware.

- Moves to and from VFP system control registers, such as FPSCR are not typically present in high-performance code, and might not be optimized. These should therefore not be placed in time-critical loops, if possible. For example, accesses to control registers on the Cortex-R7 processor are serializing, and will have a significant performance impact if used in tight loops or performance-critical code.
- Register transfer between the integer processor register bank and the floating-point register bank should similarly be avoided in time-critical loops.
- Load/store multiple operations are preferred to the use of multiple, individual floating-point loads and stores, to make efficient use of available transfer bandwidth.

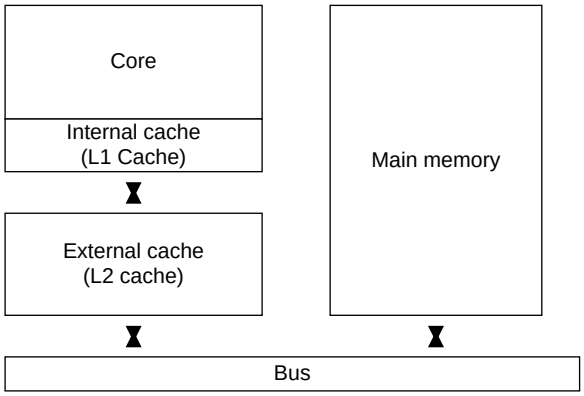
8. Caches

Essentially, a processor cache is a small, fast block of memory that sits between the core and main memory. It holds copies of recently accessed items in main memory. Accesses to the cache memory happen significantly faster than those to main memory. As the cache holds only a subset of the contents of main memory, it must store both the address of the item in main memory and the associated data. Whenever the core wants to read or write a particular address, it will first look for it in the cache. Should it find the address in the cache, it will use the data in the cache, rather than having to perform an access to main memory.

This significantly increases the potential performance of the system, by reducing the effect of slow external memory access times. It also reduces the power consumption of the system, by avoiding the requirement to drive external signals. Cortex-R series processors possess an alternative fast access memory in the form of Tightly Coupled Memory.

When the ARM architecture was first developed, the clock speed of the processor and the access speeds of memory were broadly similar. Processor cores today are much more complicated and can be clocked orders of magnitude faster. However, the frequency of the external buses and of memory devices has not scaled to the same extent. It is possible to implement small blocks of on-chip SRAM that can operate as fast as the core. But such RAM is very expensive in comparison to standard DRAM blocks, that can have thousands of times greater capacity. In many ARM processor-based systems, access to external memory will take tens or even hundreds of core cycles.

Figure 8-1: A basic cache arrangement



Cache sizes are small relative to the overall memory used in the system. Larger caches make for more expensive chips. In addition, making an internal core cache larger can potentially limit the maximum speed of the core. Significant research has gone into identifying how hardware can determine what it should keep in the cache. Efficient use of this limited resource is a key part of writing efficient applications to run on a core.

Caches speed up execution because program execution is not random. Programs often access the same set of data repeatedly and execute the same set of instructions repeatedly. By moving code or data into faster memory during their first access, subsequent accesses to that code or

data become much faster. The initial access that provided the data to the cache is no faster than normal. Because of these subsequent accesses to the cached values the performance increases. The processor hardware checks all instruction fetches, and data reads or writes in the cache. As the cache holds only a subset of main memory, there has to be a way to determine whether the required address is in the cache quickly.

On-chip SRAM can be used to implement caches, that hold temporary copies of instructions and data from main memory. Code and data have the properties of temporal and spatial locality. This means that programs tend to re-use the same addresses over time (temporal locality) and tend to use addresses that are near to each other (spatial locality). Code, for instance, can contain loops, meaning that the same code gets executed repeatedly or a function can be called multiple times. Data accesses (for example, to the stack) can be limited to small regions of memory. Access to RAM by the core exhibits such locality and is not truly random. This enables caches to be very effective.

The write buffer is a block that decouples processor writes being done by the core when executing store instructions from the external memory bus. The core places the address, control, and data values associated with the store into a set of hardware buffers. This is the write buffer. Like the cache, it sits between the core and main memory. This enables the core to move on and execute the next instruction without the requirement to stop and wait for the main memory to actually complete the write operation.

8.1 Cache drawbacks

Caches and write buffers are seen as a benefit as they speed up program execution. However, they also add some problems that are not present in an uncached core. One such drawback is that program execution time can become non-deterministic.

What this means is that, because the cache is small and holds only a subset of the main memory, it fills rapidly as a program executes. When the cache is full, existing code or data is replaced, to make room for new items. So at any given time, it is not normally possible for an application to be certain whether or not a particular instruction or data item is in the cache.

This means that the execution time of a particular piece of code can vary significantly. This can be something of a problem in hard real-time systems where strongly deterministic behavior is required. As a result, you will more than likely require a way to control how different parts of memory are accessed by the cache and write buffer.

In some cases, you want the core to read up-to-date data from an external device, such as a peripheral. It would not be sensible to use a cached value of a timer peripheral, for example. Sometimes you want the core to stop and wait for a store to complete. So, caches and write buffers give you some extra work to do.

Occasionally the contents of cache and external memory might not be the same, this is because in some caching modes, the processor can update the cache contents, that have not yet been written back to main memory. Alternatively, an agent might update main memory after a core has taken its own copy. This is a problem of coherency. This can be a problem when there are multiple cores or memory agents like an external DMA controller.

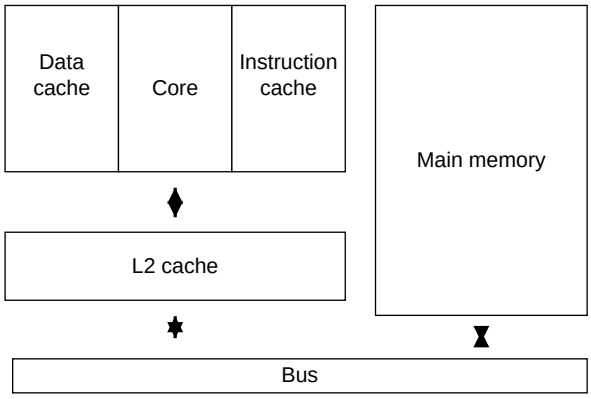
8.2 Memory hierarchy

In computer science, a memory hierarchy refers to a hierarchy of memory types, with faster and smaller memories closer to the core and slower and larger memory further away. In most systems, you can have secondary storage, such as disk drives and primary storage such as flash, SRAM and DRAM. In embedded systems, this is typically sub-divided into on-chip and off-chip memory. Memory that is on the same chip (or at least in the same package) as the core will typically be much faster.

A cache can be included at any level in the hierarchy and should improve system performance where there is an access time difference between different parts of the memory system.

The Cortex-R processors have level 1 (L1) caches, connected directly to the core logic that fetches instructions. The caches handle instruction fetches, and load and store instructions. These are Harvard caches, so there are separate caches for instructions and for data.

Figure 8-2: Typical Harvard cache



Over the years, the size of L1 caches has increased, because of SRAM size and speed improvements. At the time of writing, 16KB or 32KB cache sizes are most common, as these are the largest RAM sizes capable of providing single cycle access at a core speed of 1GHz or more.

Cortex-R series processors have an interface to an external level 2 (L2) cache. This is larger than the L1 cache (typically 256KB, 512KB or 1MB), but slower and unified (holding both instructions and data). The ARM L2C-310 is an example of such an external L2 cache controller block.

In addition, cores can be implemented in clusters where each core has its own cache. Such systems require mechanisms to maintain coherency between caches, so that when one core changes a memory location, that change is made visible to other cores sharing that memory.

8.3 Cache architecture

In a von Neumann architecture, a single cache is used for instruction and data (a unified cache). A modified Harvard architecture has separate instruction and data buses, which leads to the existence of two caches, an instruction cache (I-cache) and a data cache (D-cache). In many ARM systems, there are distinct instruction and data level 1 caches backed by a unified level 2 cache.

The cache must hold an address, data, and status information. The top bits of the 32-bit address tells the cache where the information came from in main memory and is known as the tag. The total cache size is a measure of the amount of data it can hold. The RAM used to hold tag values is not included in the calculation. The tag does, however, take up physical space in the cache.

It would be inefficient to hold one word of data for each tag address, so we typically group several locations together under the same tag. This logical block is commonly known as a cache line. The middle bits of the address, or index, identify the line. The index is used as address for the cache RAMs and does not require to be stored as a part of the tag. A cache line is said to be valid when it contains cached data or instructions, and invalid when it does not.

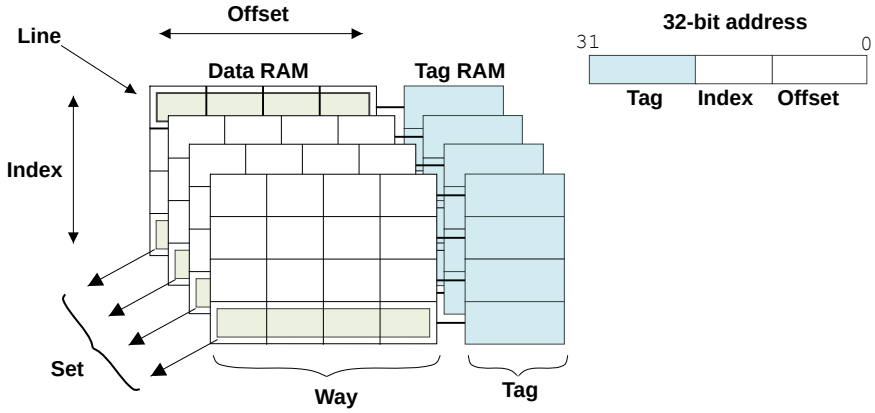
This means that the bottom few bits of the address, the offset, are not required to be stored in the tag. The processor stores the address of a line, not of each byte within the line. So the five or six least significant bits will always be 0.

Associated with each line of data are one or more status bits. Typically, there is a valid bit, marking the line as containing data that can be used. This means that the address tag represents some real value. In a data cache there is also one or more dirty bits that mark whether the cache line, or part of it, holds data that is not the same as the contents of main memory.

8.3.1 Cache terminology

A brief summary of some of the terms used might be helpful:

Figure 8-3: Cache terminology



- A line refers to the smallest loadable unit of a cache, a block of contiguous words from main memory.
- The index is the part of a memory address that determines in which line of the cache the address can be found.
- A way is a subdivision of a cache, each way being of equal size and indexed in the same fashion. The line associated with a particular index value from each cache way grouped together forms a set.
- The tag is the part of a memory address stored within the cache that identifies the main memory address associated with a line of data.

8.3.2 Direct mapped caches

We now look at various different ways of implementing caches starting with the simplest, a direct mapped cache.

In a direct mapped cache, each location in main memory maps to a single location in the cache. However, as main memory is many times larger than the cache, many addresses will map to the same cache location.

Figure 8-4: Direct mapped cache operation

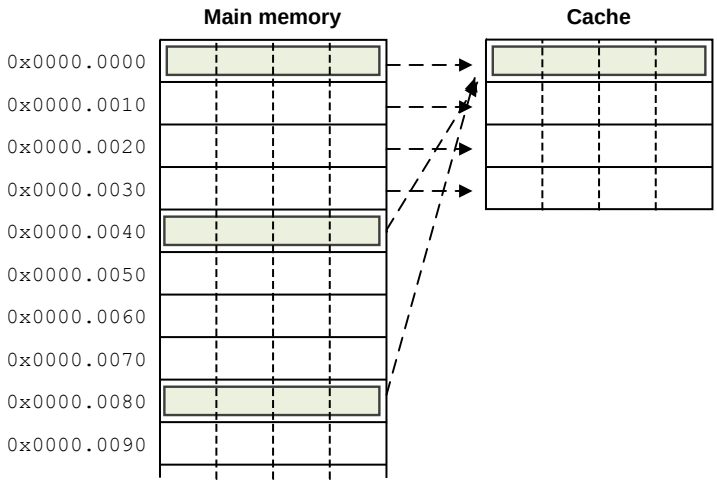
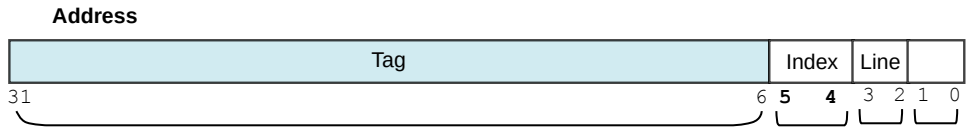


Figure 8-4: Direct mapped cache operation on page 84 shows a small cache, with four words per line and four lines. This means that the cache controller will use two bits of the address (bits [3:2]) as the offset to select a word within the line and two bits of the address (bits [5:4]) as the index to select one of the four available lines. The remaining bits of the address (bits [31:6]) is stored as a tag value.

Figure 8-5: Cache address



To look up a particular address in the cache, the hardware extracts the index bits from the address and reads the tag value associated with that line in the cache. If the two are the same and the valid bit indicates that the line contains valid data, it has a hit. It can then extract the data value from the relevant word of the cache line, using the offset and byte portion of the address. The cache does not generate a hit if the tag shows that the cache holds a different address in main memory. If the line contains valid data, but does not generate a hit then the cache line is removed and is replaced by data from the requested address.

It should be clear that all main memory addresses with the same value of bits [5:4] will map to the same line in the cache. Only one of those lines can be in the cache at any given time. This means that you can easily get a problem called thrashing. Consider a loop that repeatedly accesses address `0x00`, `0x40`, and `0x80` as in the code below:

```
void add_array(int *data1, int *data2, int *result, int size)
{
     int i;
     for (i=0 ; i<size ; i++) {
         result[i] = data1[i] + data2[i];
     }
}
```

In this code example, if `result`, `data1`, and `data2` are pointers to `0x00`, `0x40`, and `0x80` respectively then this loop will cause repeated accesses to memory locations that all map to the same line in the basic cache, as shown in Figure 8-4: Direct mapped cache operation on page 84.

At the first read of address `0x40`, it will not be in the cache and so a linefill takes place putting the data from `0x40` to `0x4F` into the cache.

Then at the read of address `0x80`, it will not be in the cache and so a linefill takes place putting the data from `0x80` to `0x8F` into the cache. And in the process the cache loses the data from address `0x40` to `0x4F`.

The result is written to `0x00`. Depending on the allocation policy this might cause another line fill. The data from `0x80` to `0x8F` might be lost.

The same thing happens on each iteration of the loop and the software runs slowly. Direct mapped caches are therefore not typically used in the main caches of ARM processors. They are used for example in the branch target address cache of the ARM1136 processor.

Processors can have hardware optimizations for situations where the whole cache line is being written to. This is a condition that can take a significant proportion of total cycle time in some systems. For example, this can happen when functions, such as `memcpy()` or `memset()`, that perform

block copies or zero initialization of large blocks are executed. In such cases, there is no benefit in first reading the data values that is over-written. This can lead to situations where the performance characteristics of the cache are different to what might normally be expected.

Cache allocate policies act as a hint to the core, they do not guarantee that a piece of memory will be read into the cache, and as a result, programmers should not rely on that.

8.3.3 Set associative caches

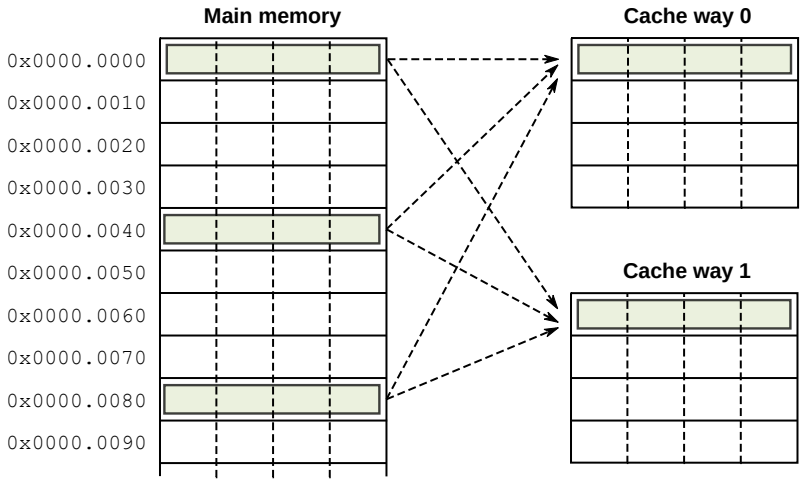
The main caches of ARM cores are always implemented using a set associative cache. This significantly reduces the likelihood of the cache thrashing seen with direct mapped caches, improving program execution speed and giving more deterministic execution. It has an increased hardware complexity and a slight increase in power use because multiple tags are compared on each cycle.

Set associative caches are divided into a number of equal sized pieces, called ways. A memory location can then map to a way rather than a line. The index field of the address continues to be used to select a particular line, but now it points to an individual line in each way. Commonly there are 2-ways or 4-ways, but some ARM implementations have higher number of ways.

External level 2 cache implementations, such as the ARM L2C-310, can have larger numbers of ways (higher associativity) because of their much larger size. The cache lines with the same index value are said to belong to a set. To check for a hit, the processor looks at each of the tags in the set.

Figure 8-6: A 2-way set-associative cache on page 86 shows a cache with 2-ways. Data from address `0x00`, `0x40`, or `0x80` might be in line 0 of either, but not both, of the two cache ways.

Figure 8-6: A 2-way set-associative cache



Increasing the associativity of the cache reduces the probability of thrashing. The ideal case is a fully associative cache, where any main memory location can map anywhere within the cache. However, building such a cache is impractical for anything other than very small caches (for

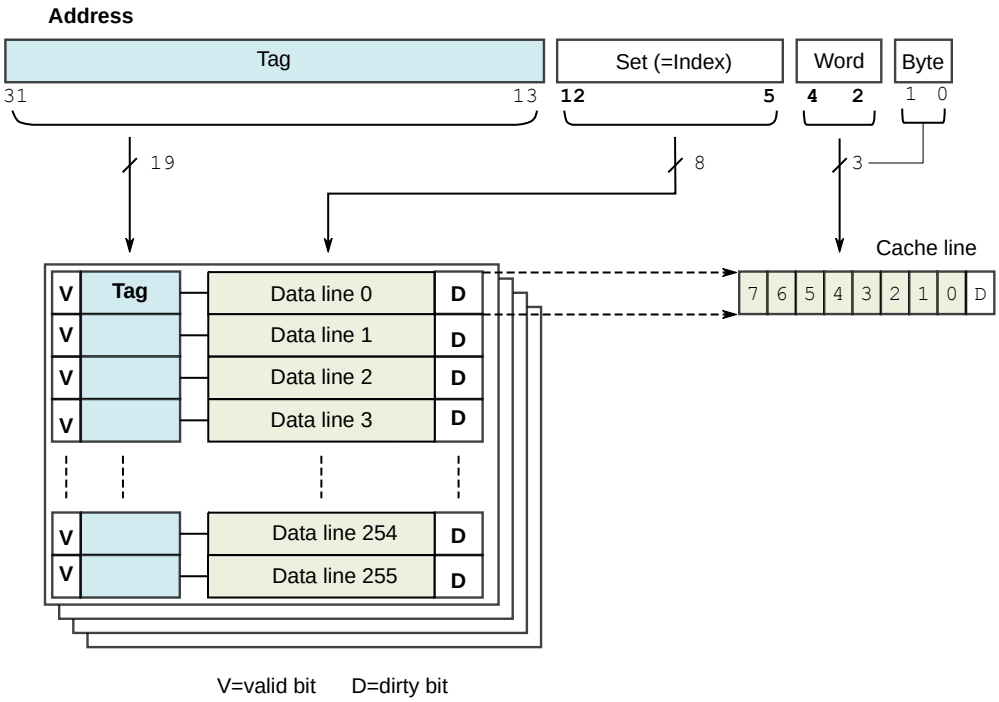
example, see The Memory Protection Unit). In practice, performance improvements are minimal for Level 1 caches above 4-way associativity, with 8-way or 16-way associativity being more useful for larger level 2 caches.

8.3.4 A real-life example

Figure 8-7: A 32KB 4-way set associative cache on page 87 is a 4-way set associative 32KB data cache, with an 8-word cache line length.

The cache line length is eight words (32 bytes) and you have 4-ways. 32KB divided by 4, divided by 32 gives a figure of 256 lines in each way. This requires eight bits to index a line within a way (bits [12:5]). Bits [4:2] of the address are used to select from the eight words within the line. The remaining bits [31:13] are used as a tag.

Figure 8-7: A 32KB 4-way set associative cache



8.3.5 Cache controller

This is a hardware block that has the task of managing the cache memory, in a way that is invisible to the program. It automatically writes code or data from main memory into the cache. It takes read and write memory requests from the core and performs the necessary actions to the cache memory or the external memory.

When it receives a request from the core it must check to see whether the requested address is in the cache. This is known as a cache look-up. It does this by comparing a subset of the address bits

of the request with tag values associated with lines in the cache. If there is a match and the line is marked valid then the read or write will happen using the cache memory.

When the core requests instructions or data from a particular address, but there is no match with the cache tags, or the tag is not valid, a cache miss results and the request must be passed to the next level of the memory hierarchy, that might be an L2 cache, or external memory. It can also cause a cache linefill. A cache linefill causes the contents of a piece of main memory to be copied into the cache. At the same time, the requested data or instructions are streamed to the core. This process happens transparently and is not directly visible to you.The core does not have to wait for the linefill to complete before using the data. The cache controller will typically access the critical word within the cache line first. For example, if a load instruction misses in the cache and triggers a cache linefill, the first read to external memory is that of the actual address supplied by the load instruction. This critical data is supplied to the processor pipeline, while the cache hardware and external bus interface then read the rest of the cache line, in the background.

8.4 Cache policies

There are a number of different choices that affect cache operation. For example:

Allocation policy
: considers what causes a line from external memory to be placed into the cache.

Replacement policy
: controls how the controller decides which line within a set associative cache to use for the incoming data.

Write policy
: controls what happens when the processor performs a write that hits in the cache.

8.4.1 Allocation policy

When the core does a cache look-up and the address it wants is not in the cache, it must determine whether or not to perform a cache linefill and copy that address from memory.

- A read allocate policy allocates a cache line only on a read. If a write is performed by the core that misses in the cache, the cache is not affected and the write goes to the next level of the hierarchy.
- A write allocate policy allocates a cache line for either a read or write that misses in the cache. And so this might more accurately be called a read-write cache allocate policy. For both memory reads that miss in the cache and memory writes that miss in the cache, a cache linefill is performed. This is typically used in combination with a write-back write policy on current ARM cores. For more information see Write policy.

8.4.2 Replacement policy

When there is a cache miss, the cache controller must select one of the cache lines in the set for the incoming data. The cache line selected is called the victim. If the victim contains valid, dirty data, the contents of that line must be written to main memory before new data can be written to the victim cache line. This is called eviction.

The replacement policy is what controls the victim selection process. The index bits of the address are used to select the set of cache lines, and the replacement policy selects the specific cache line from that set that is to be replaced.

Most ARM processors support:

- Round-robin or cyclic replacement means that there is a counter, called the victim counter, that cycles through the available ways and cycles back to 0 when it reaches the maximum number of ways.
- Pseudo-random replacement randomly selects the next cache line in a set to replace. The victim counter is incremented in a pseudo-random fashion and can point to any line in the set.

The Cortex-R4, Cortex-R5, and Cortex-R7 processors only support the pseudo-random policy.

A round-robin replacement policy is generally more predictable, but can suffer from poor performance in certain use cases and for this reason, the pseudo-random policy is often preferred.

8.4.3 Write policy

When the core executes a store instruction, a cache lookup on the address to be written is performed. For a cache hit on a write, there are two choices.

- Write-through. With this policy writes are performed to both the cache and main memory. This means that the cache and main memory are kept coherent. As there are more writes to main memory, a write-through policy is slower than a write-back policy if the write buffer fills. Therefore write-through is less commonly used, although it can be useful for debug. Regions marked as write-through are treated as non-cacheable.

  The Cortex-R4 and Cortex-R5 processors use write-through when RAM parity protection is enabled. The Cortex-R5 processor also uses write-through so that the Micro Snoop Control Unit (µSCU) can maintain coherency between the L1 cache and the master connected to the Accelerator Coherency Port (ACP).
- Write-back. In this case, writes are performed only to the cache, and not to main memory. This means that cache lines and main memory can contain different data. The cache line holds newer data, and main memory contains older data (said to be stale). To mark these lines, each line of the cache has an associated dirty bit (or bits). When a write happens that updates the cache, but not main memory, the dirty bit is set. If the cache later evicts a cache line whose dirty bit is set (a dirty line), it writes the line out to main memory. Using a write-back cache policy can significantly reduce traffic to slow external memory and therefore improve performance and save power. However, if there are other agents in the system that can access memory at the same time as the processor, there might be coherency issues.

The Cortex-R7 processor only supports the write-back policy.

8.4.4 Choosing the best write policy

System designers should evaluate the policy for cache operation best suited to their requirements. If a write-back policy is used then the cache will often have to be cleaned as described before switching context so that coherency in the memory system is maintained. This can require lots of writes to CP15 registers prior to switching. Alternatively, choosing a write-through policy can reduce system performance and increase power consumption as coherency is maintained on every cache operation, which is sometimes unnecessary. However, this means that the cache is being cleaned continuously and so cache maintenance will typically take less time.

8.5 Write and Fetch buffers

A write buffer is a hardware block inside the core implemented using a number of buffers. Sometimes it is present in other parts of the system as well. It accepts address, data, and control values associated with core writes to memory. When the core executes a store instruction, it might place the relevant details, such as the location to write to, the data to be written, and the transaction size into the buffer. The core does not have to wait for the write to be completed to main memory. It can proceed executing the next instruction. The write buffer itself will drain the writes accepted from the core, to the memory system.

A write buffer can increase the performance of the system because the core does not have to wait for stores to complete. In effect, provided there is space in the write buffer, the write buffer is a way to hide latency. If the number of writes is low or well spaced, the write buffer will not become full. If the core generates writes faster than they can be drained to memory, the write buffer will eventually fill and there is little performance benefit.

Some write buffers support write merging, also called write combining. They can take multiple writes, for example, a stream of writes to adjacent bytes, and merge them into one single burst. This can reduce the write traffic to external memory and therefore boost performance.

It will be obvious to the experienced programmer that sometimes the behavior of the write buffer is not as expected. When accessing a peripheral you might want the core to stop and wait for the write to complete before proceeding to the next step. Sometimes you might want a stream of bytes to be written and do not want the stores to be combined. ARM memory ordering model, looks at memory types supported by the ARM architecture and how to use these to control how the caches and write buffers are used for particular devices or parts of the memory map.

Similar components, called fetch buffers, can be used for reads in some systems. In particular, cores typically contain prefetch buffers that read instructions from memory ahead of them actually being inserted into the pipeline. In general, such buffers are transparent to you. Some possible hazards associated with this will be considered when we look at memory ordering rules

8.6 Cache performance and hit rate

The hit rate is defined as the number of cache hits divided by the number of memory requests made to the cache during a specified time, normally calculated as a percentage. Similarly, the miss rate is the number of total cache misses divided by the total number of memory requests made to the cache. You can also calculate the number of hits or misses on reads or writes only.

Clearly, a higher hit rate will generally result in higher performance. It is not really possible to quote example figures for typical software, the hit rate is very dependent on the size and spatial locality of the critical parts of the code or data operated on and of course, the size of the cache.

There are some simple rules that can be followed to give better performance. The most obvious of these is to enable caches and write buffers and to use them wherever possible. The rules can be used for all parts of the memory system that contain code, and more generally for RAM and ROM, but not peripherals. Performance is considerably increased if instruction memory is cached. Placing frequently accessed data together in memory can also be helpful. For example, a frequently accessed array will benefit from having a base address at the start of a cache line.

Fetching a data value in memory involves fetching a whole cache line. If none of the other words in the cache line is used, there is little or no performance gain. Smaller code might cache better than larger code and this can sometimes give paradoxical results. For example, a piece of C code might fit entirely within the cache when compiled for Thumb, or for the smallest size, but not when compiled for ARM or for maximum performance. As a consequence it can run faster than the more optimized version.

8.7 Invalidating and cleaning cache memory

Cleaning and invalidation can be required when the contents of external memory have been changed and you want to remove stale data from the cache. It can also be required after MPU related activity such as changing access permissions or cache policies.

The word flush is often used in descriptions of clean and invalidate operations. ARM generally uses only the terms clean and invalidate.

- Invalidation of a cache or cache line means to clear it of data. This is done by clearing the valid bit of one or more cache lines. The cache must always be invalidated after reset as its contents are undefined. If the cache contains dirty data, it is generally incorrect to invalidate it. Any updated data in the cache from writes to write-back cacheable regions would be lost by simple invalidation.
- Cleaning a cache or cache line means writing the contents of dirty cache lines out to main memory and clearing the dirty bits in the cache line. This makes the contents of the cache line and main memory coherent with each other. This is only applicable for data caches in which a write-back policy is used.

  Cache invalidate, and clean operations can be performed by cache set, or way, or by specifying a particular address.

Self-modifying code, or copying code from one location to another, might mean you have to clean or invalidate the cache. The memory copy code will use load and store instructions and these will operate on the data side of the processor. If the data cache is using a write-back policy for the area to which code is written, it is necessary to clean that data from the cache before the code can be executed. This ensures that the instructions stored as data go out into main memory and are then available for the instruction fetch logic. In addition, if the area to which code is written was previously used for some other program, the instruction cache could contain stale code from before main memory was re-written. Therefore, it might also be necessary to invalidate the instruction cache before branching to the newly copied code.

The commands to either clean or invalidate the cache are CP15 operations. They are available only to privileged code and cannot be executed in User mode.

CP15 instructions exist that will clean, invalidate, or clean and invalidate level 1 data or instruction caches. Invalidation without cleaning is safe only when it is known that the cache cannot contain dirty data, for example a Harvard instruction cache. You can perform the operation on the entire cache, or on individual lines. These individual lines can be specified either by giving the address to be cleaned or to be invalidated, or by specifying a line number in a particular set, in cases where the hardware structure is known. The same operations can be performed on the L2 or outer caches. For more information see Level 2 cache controller. A typical example of such code can be found in Booting a bare-metal system.

A common situation where cleaning or invalidation can be required is Direct Memory Access (DMA). When it is required to make changes made by the processor visible to external memory, so that it can be read by a DMA controller, it might be necessary to clean the cache. When external memory is written by a DMA controller and it is necessary to make those changes visible to the processor, the affected addresses must be invalidated in the cache.

```
setup_caches
      MRC p15, 0, r1, c1, c0, 0           ; Read System Control Register (SCTLR)
      BIC r1, r1, #1                      ; mpu off
      BIC r1, r1, #(1 << 12)              ; i-cache off
      BIC r1, r1, #(1 << 2)               ; d-cache & L2-$ off
      MCR p15, 0, r1, c1, c0, 0           ; Write System Control Register (SCTLR)
 ;-----------------------------------------------
 ; 1.MPU, L1$ disable
 ;-----------------------------------------------
      MRC p15, 0, r1, c1, c0, 0           ; Read System Control Register (SCTLR)
      BIC r1, r1, #1                      ; mpu off
      BIC r1, r1, #(1 << 12)              ; i-cache off
      BIC r1, r1, #(1 << 2)               ; d-cache & L2-$ off
      MCR p15, 0, r1, c1, c0, 0           ; Write System Control Register (SCTLR)
 ;-----------------------------------------------
```

```
; 2. invalidate: L1$, branch predictor
;----------------------------------------------
     MOV     r0, #0
     MCR     p15, 0, r0, c7, c5, 0       ; Invalidate Instruction Cache
     MCR     p15, 0, r0, c7, c5, 6       ; Invalidate branch prediction array
     ISB                                 ; Instruction Synchronization Barrier
;----------------------------------------------
; 2.a. Enable I cache + branch prediction
;----------------------------------------------
     MRC     p15, 0, r0, c1, c0, 0           ; System control register
     ORR     r0, r0, #1 << 12                ; Instruction cache enable
     ORR     r0, r0, #1 << 11                ; Program flow prediction
     MCR     p15, 0, r0, c1, c0, 0           ; System control register
;----------------------------------------------
```

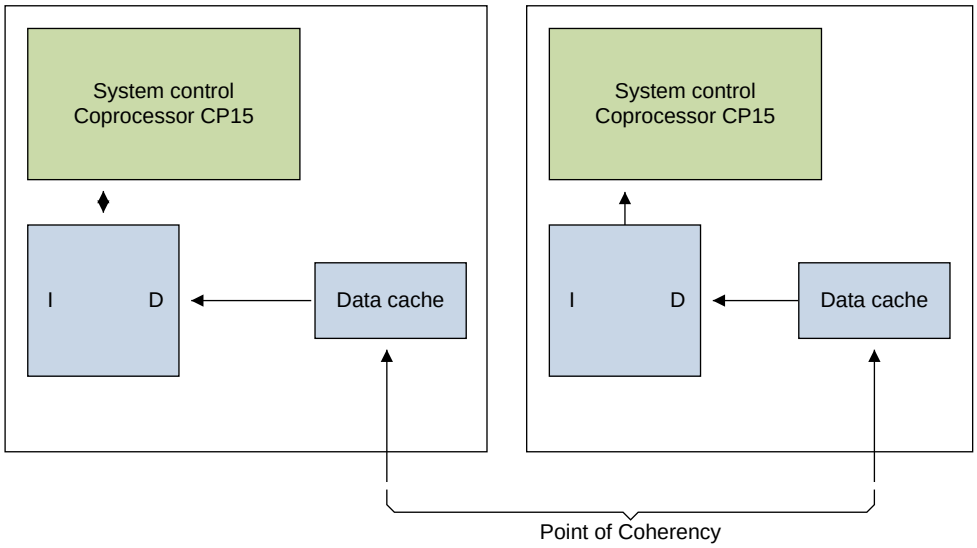
8.8 Point of coherency and unification

For set-based or way-based clean and invalidate, the operation is performed on a specific level of cache. The architecture defines two conceptual points for operations that use an address.

Point of Coherency (PoC)

For a particular address, the PoC is the point at which all blocks, for example, cores, DSPs, or DMA engines, that can access memory are guaranteed to see the same copy of a memory location. Typically, this is the main external system memory.

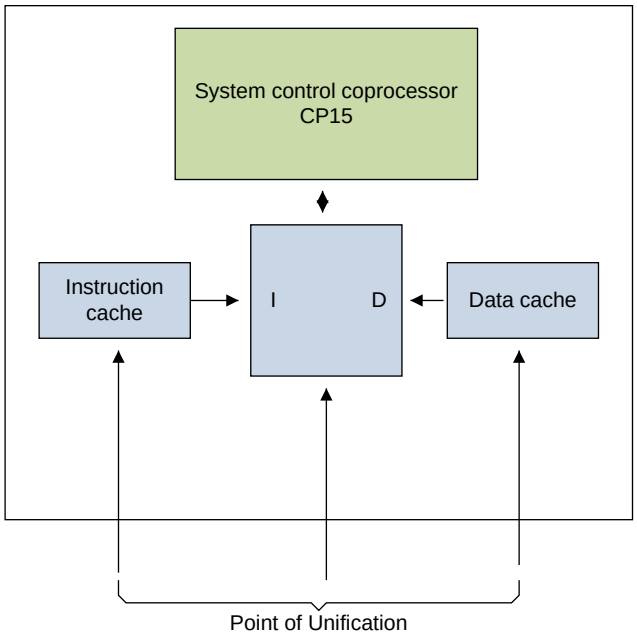
Figure 8-8: Point of Coherency



Point of Unification (PoU)

The PoU for a core is the point at which the instruction and data caches of the core are guaranteed to see the same copy of a memory location. If no external cache is present, main memory would be the Point of Unification.

Figure 8-9: Point of Unification



If no external cache is present, main memory would be the PoU.

Knowledge of the PoU enables self-modifying code to ensure future instruction fetches are correctly made from the modified version of the code. They can do this by using a two-stage process:

1. Clean the relevant data cache entries (by address).
2. Invalidate instruction cache entries (by address).

In addition, the use of memory barriers is required.

8.8.1 Example code for cache maintenance operations

The following code illustrates a generic mechanism for cleaning the entire data or unified cache to the point of coherency.


Note

In the case of a cluster where multiple cores share a cache before the point of coherency, running this sequence on multiple cores results in the operations being repeated on the shared cache

```
     MRC p15, 1, R0, c0, c0, 1   ; Read CLIDR into R0
     ANDS R3, R0, #0x07000000
     MOV R3, R3, LSR #23         ; Cache level value (naturally aligned)
     BEQ Finished
     MOV R10, #0

Loop1
     ADD R2, R10, R10, LSR #1    ; Work out 3 x cachelevel
     MOV R1, R0, LSR R2          ; bottom 3 bits are the Cache type for this level
     AND R1, R1, #7              ; get those 3 bits alone
     CMP R1, #2
     BLT Skip                    ; no cache or only instruction cache at this level
     MCR p15, 2, R10, c0, c0, 0  ; write CSSELR from R10
     ISB                         ; ISB to sync the change to the CCSIDR
     MRC p15, 1, R1, c0, c0, 0   ; read current CCSIDR to R1
     AND R2, R1, #7              ; extract the line length field
     ADD R2, R2, #4              ; add 4 for the line length offset (log2 16 bytes)
     LDR R4, =0x3FF
     ANDS R4, R4, R1, LSR #3     ; R4 is the max number on the way size (right
 aligned)
     CLZ R5, R4                  ; R5 is the bit position of the way size increment
     MOV R9, R4                  ; R9 working copy of the max way size (right
 aligned)

Loop2
     LDR R7, =0x00007FFF
     ANDS R7, R7, R1, LSR #13    ; R7 is the max num of the index size (right
 aligned)

Loop3
     ORR R11, R10, R9, LSL R5    ; factor in the way number and cache number into
 R11
     ORR R11, R11, R7, LSL R2    ; factor in the index number
     MCR p15, 0, R11, c7, c10, 2 ; DCCSW, clean by set/way
     SUBS R7, R7, #1             ; decrement the index
     BGE Loop3
     SUBS R9, R9, #1             ; decrement the way number
     BGE Loop2

Skip
```

```
     ADD R10, R10, #2            ; increment the cache number
     CMP R3, R10
     BGT Loop1
     DSB

Finished
```

8.9 Level 2 cache controller

The Cortex-R series processors do not have an integrated level 2 cache. However, the system designer can connect the ARM L2 cache controller (L2C-310) outside of the processor instance.

The L2C-310 cache controller can support a cache of up 8MB in size, with a set associativity of between four and sixteen ways. The size and associativity are fixed by the SoC designer. The level 2 cache can be shared between multiple processors, or between the processor and other agents, for example a graphics processor. It is possible to lockdown cache data on a per-master per-way basis, enabling management of cache sharing between multiple components.

8.9.1 Level 2 cache maintenance

You might at some point have to clean or invalidate some or all of an external cache. This can be done by writing to memory-mapped registers within the L2 cache controller in the case where the cache is external to the processor, or through CP15, where the level 2 cache is implemented inside the processor. The registers themselves are not cached, that makes this feasible. Where such operations are performed by having the processor perform memory-mapped writes, the processor requires a way of determining when the operation is complete. It does this by polling a memory-mapped register within the L2 cache controller.

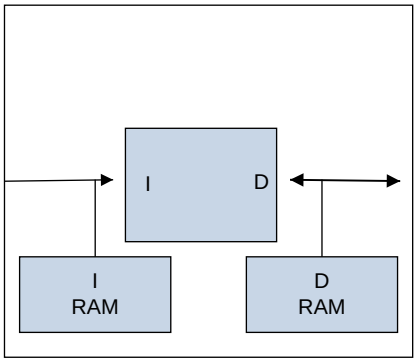
9. Tightly Coupled Memory

Tightly Coupled Memory (TCM) provides low-latency memory accesses that the core can use without the unpredictability of access time that is a feature of caches. When using external, cacheable memory a requested instruction or piece of data might be in the cache, giving a fast access, or might not be in the cache, requiring a slower access to external memory. When using TCM the access time is consistent.

The TCM can be used to hold time-critical routines, such as interrupt handling routines or real-time tasks where the indeterminacy of a cache is undesirable. In addition, you can use it to hold ordinary variables, data types whose locality properties are not well suited to caching, and critical data structures such as interrupt stacks.

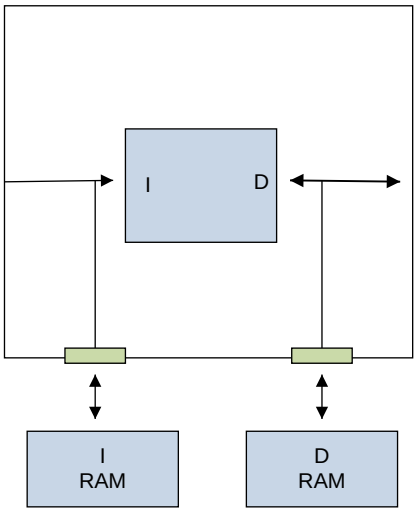
A TCM is physically located very close to the processor core. Accesses to the TCM will typically be configured to capture or return data in a single cycle. By storing time-critical routines such as exception handlers in the TCM, the processor can have immediate access to the sub-routine rather than having to wait for an initial code fetch from external memory.

Figure 9-1: Instruction and data TCMs integrated into the core



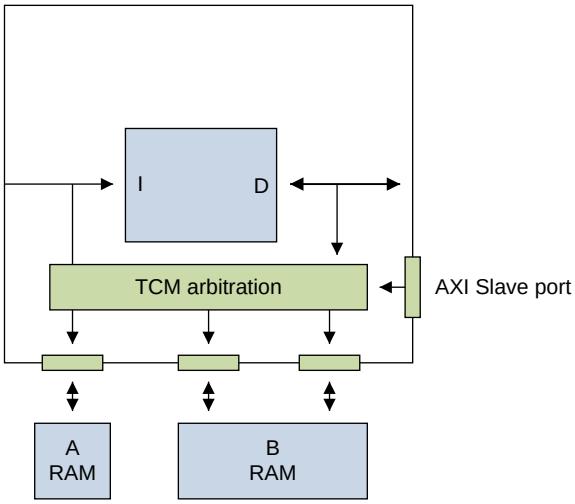
In Figure 9-1: Instruction and data TCMs integrated into the core on page 97 the instruction and data TCMs have been integrated into the core. There are separate paths to each TCM, one for fetching instructions and one for loading or storing data. These paths are independent and can both be active at the same time. Both code and data can be copied to the TCMs by application or library code.

Figure 9-2: Instruction and data TCMs independent of the core



In Figure 9-2: Instruction and data TCMs independent of the core on page 98 the instruction and data TCMs are independent of the processor core. The processor core has two ports, one for instructions and one for data. Integrating the TCMs into the processor core allows the implementation to be optimized for performance. Implementing the TCMs externally to the core allows the implementation greater flexibility but might reduce the maximum performance.

Figure 9-3: TCM arbitration



In Figure 9-3: TCM arbitration on page 98 there are two external TCMs that can be configured to only store instructions, only data, or a mixture of the two. Enabling a TCM to include both instructions and data provides more flexibility from a system perspective but might limit performance compared with optimizing a TCM to solely store instructions or data. The BTCM is accessible via two ports. This indicates that the TCM has been implemented as two separate banks of RAM so that the two banks can be accessed simultaneously.

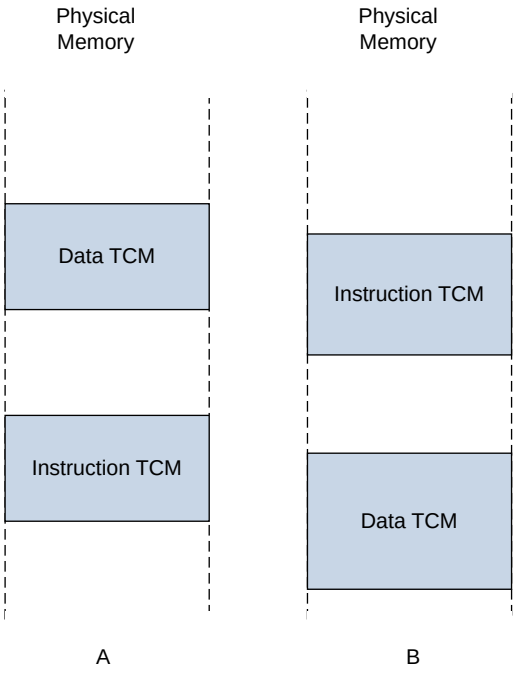
As well as the processor core these two TCM instances are also accessible from a slave port. The presence of the slave port means that the TCMs can be preloaded with values: instruction code can be loaded into the TCMs directly from the slave port after system power up and the processor can then fetch instructions directly from the TCMs without having to access external memory.

Accesses from the processor core, the instruction fetches and data loads and data stores, and accesses from the slave will need to be arbitrated internally. Separate accesses to the ATCM and BTCM could take place simultaneously but when there is contention for access to a TCM then the internal arbitration will need to prioritize accesses from different sources. Typically the data accesses, loads and store, will have highest priority, then instruction fetches and accesses from the slave port will have the lowest priority.

9.1 Location of the TCM in the memory map

A TCM is used as part of the physical memory map of the system, and, unlike a cache, does not have to be backed by a level of external memory with the same physical addresses. Each TCM has a dedicated base address, and typically the TCMs can be positioned anywhere within the 32-bit address space, at any naturally aligned address, with the proviso that separate TCM regions must not overlap with each other and that each region is located on an address boundary which is a multiple of its size. In Figure 9-4: Location of TCM in physical memory on page 99, both A and B are equally valid locations. TCM location can be changed at run time through CP15.

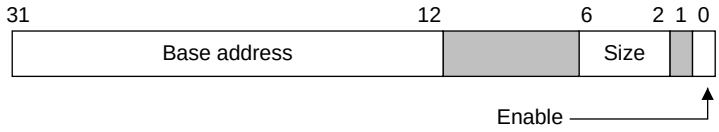
Figure 9-4: Location of TCM in physical memory



Accesses are sent directly to the TCM interface with no accesses required on the external memory bus.

Anything in the external memory at the same address locations as the TCM is not accessible to the processor when the TCM is enabled. This means that you have to be careful when enabling or disabling a TCM or when re-locating a TCM in the memory map. You have to ensure that all memory accesses go where they are intended, either to external memory or to the TCM. Memory barriers must therefore be used before and after enabling or disabling the TCMs.

Figure 9-5: TCM configuration



When a block of TCM memory is enabled, it will exist as memory in the physical memory map. If external memory already exists at this physical address, the TCM memory will have priority and any accesses to that memory range will go to TCM.

The size of the TCM is set by the implementation. However it is possible to control the location of the TCM in the processor memory map. It is also possible for privileged code running on the processor to enable and disable the TCM. The TCMs can also be configured to be enabled out of reset at a particular address location. Enabling or disabling the TCMs does not act as a data or instruction memory barrier. You must take care of this yourself. One example of when you might want use these operations is shown below.

```
DSB                                 ; Data synchronization barrier
MOV r0, #0x800000001                  ; Set base address, with enable bit set
MCR p15, 0, r0, c9, c1, 0           ; Write TCM region register
ISB                                 ; Instruction synchronization barrier
```

9.2 Performance of TCM compared to cache

In certain situations, TCM performance is better than cache performance. In other situations, cache performance is better than TCM performance.

If an instruction or data fetch misses in the cache, the processor fetches from the external memory. In this case the performance of the TCM is significantly better than the cache. This is because the cache miss requires an external memory fetch and allocation of data into the cache.

However, if the instructions and data have been pre-fetched from external memory, populating the processor caches, then for subsequent fetches of the cached instructions and data, the cache performance compared to TCM performance is different:

- For data fetches, TCM performance will generally be comparable to cache performance, when there is a cache hit.
- For instruction fetches, TCM performance is a little lower than cache performance.

When instructions include literal pool accesses, the processor must fetch a constant value from memory to interpret the instruction. In this situation cache performs faster than TCM:

- If the instruction has a hit in the instruction cache then the processor fetches the literal pool value from the data cache. This leaves the instruction cache interface available to fetch additional instructions.
- If the instructions are stored in a TCM then the processor must access the TCM interface twice:
  - To fetch the original instruction.
  - To fetch the literal pool value.

It might be tempting to make the TCMs as large as possible to benefit from the faster access time for as much code as possible, however this might not provide the best system performance.

As the size of the TCM is increased, it becomes more difficult to meet the timing requirements of the design during the implementation on silicon. This can limit the performance of the processor.

Also, large areas of SRAM local to the processor is more expensive from a silicon manufacturing perspective compared with the cost of using external memory devices.

Another consideration is that the TCMs are local to the processor and so the instructions and data are non-shareable. Instructions and data that are shareable must be placed in common areas of memory.

9.3 Loading values into TCMs

It is common in Cortex-R series processors to store exception handler code in a TCM. The first exception routine encountered is the reset handler. Ideally the processor then fetches the reset handler routine from the TCM. However, when the processor is powered up, the TCM is uninitialized. There are three possible solutions to this problem:

- Define part of or the whole of the TCM as non-volatile memory and hard-code the exception handler routines. However, this is not a practical solution.
- Store the initial reset handler in external memory.
- When the system powers up, the processor fetches the initial reset handler from external memory. The processor then copies the exception handlers from external memory into the TCM and relocates the TCM to the vector table location. The relocation step is necessary because the external memory and the TCM cannot appear at the same location in the physical memory map when the memory copying is in progress.
- Use an external device to copy data from the external memory into the TCM before the processor starts to fetch the instructions.
- Cortex-R series processors include an AXI slave interface that enables an external bus master to write and read from the TCMs. When the processor comes out of reset it is prevented from fetching instruction code by the assertion of a control signal. The external master can then pre-load the TCM instances with the instructions from external memory. When the preload is complete the control signal is released and the processor can fetch the reset handler from the TCM.

9.4 TCM Properties in the Cortex-R4 and Cortex-R5 processors

The TCMs in the Cortex-R4 and Cortex-R5 processors are separate from the main processor. The reason for this is to increase implementation flexibility.

Each TCM can be up to 8 MB in size. The TCMs can run at full processor speed or the TCMs can delay accesses by one or more cycles. This can improve the timing closure of the TCMs, especially for larger TCMs, at the expense of one or more additional cycles of latency. The TCMs can be implemented as SRAM or ROM.

The TCMs can also be implemented with error detection and correction features. For more information, see Fault Detection and Control Features. The processor can be pin configured to enable one or both of the TCMs when the processor comes out of reset. At reset, one of the TCMs is located at address `0x00000000` and the other TCM is located at an implementation defined address. Asserting the nCPUHALT pin while the processor is in reset prevents the processor from fetching instructions immediately out of reset. The TCM located at address `0x00000000` would then be preloaded using the AXI slave interface. When the nCPUHALT signal is released the processor will fetch instructions directly from the TCM.

Accesses to the TCM are controlled by the MPU though the TCM itself must have the properties of normal memory. In addition, Device or Strongly-ordered memory in the MPU has the Execute-Never (XN) property, and it would not be possible to fetch instructions from TCM address ranges which have accidentally been made Device or Strongly-ordered.

9.5 TCM properties in the Cortex-R7 processor

The TCMs in the Cortex-R7 processor are fully integrated into the design to optimize performance.

There are separate TCM instances for instruction code and data. Each TCM can be up to 128 KB in size.

The data TCM has to run at the full processor speed whereas the instruction TCM can be configured to delay accesses by a single cycle. This can improve timing closure of the instruction TCM at the expense of an extra cycle of latency when fetching instructions. The TCMs must be implemented as SRAM, not ROM.

The TCMs can also be implemented with error detection and correction features. For more information, see Fault Detection and Control Features.

The processor can be pin configured to enable the instruction TCM when the processor comes out of reset. Asserting the nCPUHALT pin while the processor is in reset prevents the processor from fetching instructions immediately out of reset. The instruction TCM would then be preloaded using the AXI slave interface. When the nCPUHALT signal is released, the processor fetches instructions directly from the TCM.

Accesses to the TCM are controlled by the MPU though the TCM itself must have the properties of normal memory.

9.6 Quality of Service

The Cortex-R7 processor has an out-of-order pipeline, in contrast to the primarily in-order pipeline of the Cortex-R4 and Cortex-R5 processors. This means that the Cortex-R7 processor is more optimized towards performance than the Cortex-R4 and Cortex-R5 processors. This also means that the Cortex-R7 processor is inherently less deterministic. However the Cortex-R7 processor also includes features to help prioritize more critical code. These features are called Quality of Service (QoS) features.

The memory subsystem of the Cortex-R7 processor is specifically designed to support these QoS features. Typically the most timing-critical code is stored in the Tightly Coupled Memories. The timing-critical peripherals are connected using the peripheral port. The Cortex-R7 processor is configured with two AXI master ports. The two master ports then connect to distinct memory subsystems and address filtering is then used to differentiate between the two subsystems.

One area of the memory map is dedicated to more time-critical code. Typically this AXI master port, master port M1, connects to a region of on-chip SRAM that can provide a considerably faster response time than off-chip memory. Finally, the least timing-critical code and peripherals are connected to the main memory subsystem, accessed using the AXI master port 0 in the Cortex-R7 processor.

Quality of Service features are then enabled to optimize the accesses into this memory system.

The Quality of Service features can be used to ensure that low priority cacheable traffic does not block the flow of accesses from:

- Peripherals connected on the peripheral port.
- Data TCM accesses.
- Cacheable traffic connected on the optional external AXI master port 1, when used with address filtering.

Transfers going through AXI master port 0, are considered to have low priority.

The QoS bit in the Auxiliary Control Register is used to enable QoS:

- If this bit is set, some hardware resources are reserved solely for high priority traffic. These resources are not accessible by low priority traffic. This means that high priority traffic, such as an Interrupt Service Routine (ISR), has the necessary resources to start executing even if low priority transactions are in progress. When the low priority traffic completes its pending transfers, the high priority traffic is then be able to use all the hardware resources.
- If this bit is not set, no hardware resources are reserved for high priority traffic, and both the low and high priority traffic share and use all the available resources. This configuration has better average performance, because all hardware resources are available to all traffic.

You can set the QoS bit on a per core basis to ensure that low priority cacheable traffic, with significant memory latencies, does not block the flow of traffic from these tasks. The SCU offers some QoS as soon as the filtering is enabled on the AXI master ports.

You can use the QoS bit to set different mixes of traffic flows:

- If the QoS bit is not set, all traffic can use all hardware resources regardless of priority.
- If the QoS bit is set, low priority traffic cannot use all the hardware resources.

9.6.1 Access to peripherals

The Cortex-R5 and Cortex-R7 processors include an additional AXI port specifically to provide access to time-critical peripherals. Only those peripherals closely coupled to the performance of the processor must be connected to this port.

For example, this port could be used to access an external interrupt controller for the Cortex-R5 processor. This might also include peripherals that generate high priority interrupts for the processor. Having the dedicated ports means that accesses to these peripherals do not contend with lower priority memory accesses in the rest of the memory subsystem.

10. The Memory Protection Unit

Many real-time systems operate with a multitasking operating system (OS). The OS provides a facility to ensure that the task currently executing does not disrupt the operation of other tasks. System resources, the code, and data of other tasks are protected. The protection system typically relies on both hardware and software.

In a system with no hardware protection support, each task must work in a cooperative way with other tasks and follow rules. In contrast, a system with dedicated protection hardware will check and restrict access to system resources, preventing hostile or unintentional access to forbidden resources. Tasks are still required to follow a set of OS rules, but these are also enforced by hardware, that gives more robust protection.

ARM provides all of the Cortex-R series processors with this capability using a Memory Protection Unit (MPU). This provides hardware protection over a number of software-programmed regions, but does not provide a full virtual memory system with address translation.

The Cortex-R series processors implement the ARM Protected Memory System Architecture (PMSA).

The PMSA is based on a Memory Protection Unit (MPU). The PMSA provides a much simpler memory protection scheme than the MMU based Virtual Memory System Architecture (VMSA). The simplification applies to both the hardware and the software.The main simplification is that the MPU does not use translation tables. Instead, the System Control Coprocessor (CP15) registers define protection regions. The protection regions eliminate the need for hardware to perform translation table walks and software to set up and maintain translation tables.The use of protection regions has the benefit of making the memory checking fully deterministic. However, the level of control is region based rather than page based, meaning the control is considerably less precise than in the VMSA.A second simplification is that the PMSA does not support virtual to physical address mapping.

In the Cortex-A series processors, a Memory Management Unit (MMU) controls access to the memory subsystem. This provides greater flexibility in controlling the memory accesses but requires that some configuration data for the MMU is stored in external memory. This means that memory access times can vary considerably when using an MMU. In contrast, all the configuration of the Memory Protection Unit (MPU) is internal to the Cortex-R processors and so the use of the MPU will not impact the memory access latency.

10.1 Memory subsystem

The use of a Memory Protection Unit (MPU) and Tightly Coupled Memories in the Cortex-R processor implementations can help ensure fast access to critical subroutines in the code. The MPU controls access to the different regions of the memory map.

The ARM MPU uses these regions to manage system protection. A region is a set of attributes associated with an area of memory. The core holds these attributes in CP15 registers and identifies each region with a number.

The memory boundaries of a region are defined by its base address and its size. Each region possesses additional attributes that define access rights, memory type and the cache policies. Because peripherals are memory-mapped in ARM systems, the same protection mechanism is used for both system peripherals and task memory.

Each region consists of:

- A base address.
- Region Size and Enable.
- Memory Type and Access Control.
- Optional sub-regions.

10.2 Implementing a Protected Memory System with Regions

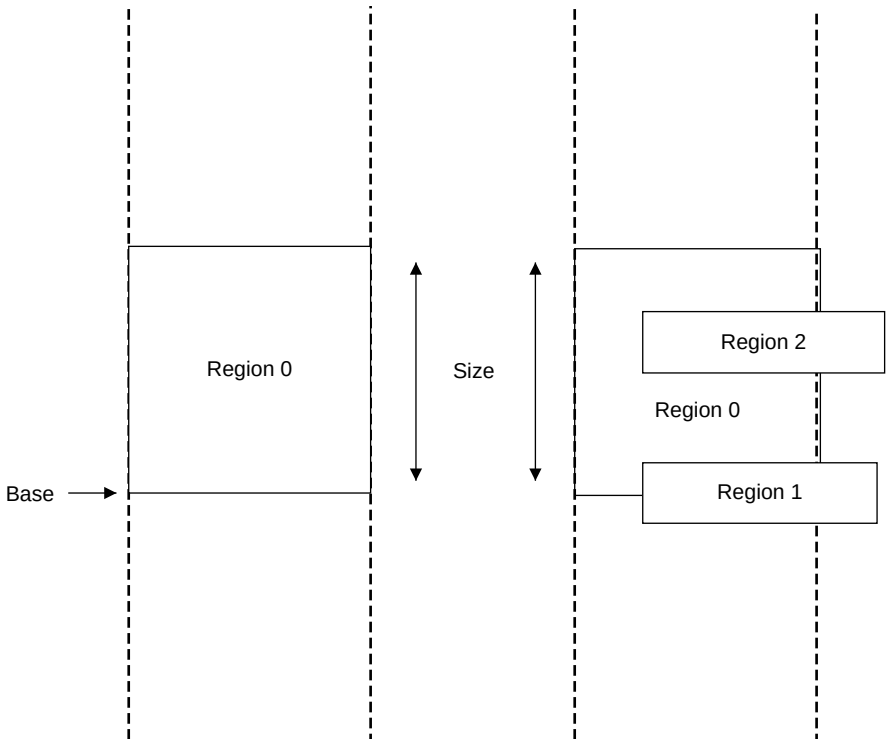
To implement a protected system, the OS must define a number of regions to cover the different areas in the main memory map. This can be done as a static (fixed) scheme, during the boot sequence, that persists while the system is running. Alternatively, more complex systems can assign (and remove) regions dynamically as tasks start and finish, or as the software context switches.

There are a number of points to consider when dealing with regions:

- Regions are assigned a priority number that is independent of the privileges assigned to the region.
- Regions can overlap other regions. In this case, the attributes of the region with the highest priority number take precedence over the other regions (only for the addresses within the areas that overlap).
- The size of a region can be any power of two between 32 Bytes and 4 GB.
- Accessing an area of memory outside of a defined region might result in an abort.

Overlapping regions provide a greater flexibility when assigning access permissions. A useful feature provided by overlapping regions is a background region. A low priority region is used to assign the same attributes to a large memory area. Other regions with higher priority are then placed over this background region to change the attributes of a smaller part of the memory map.

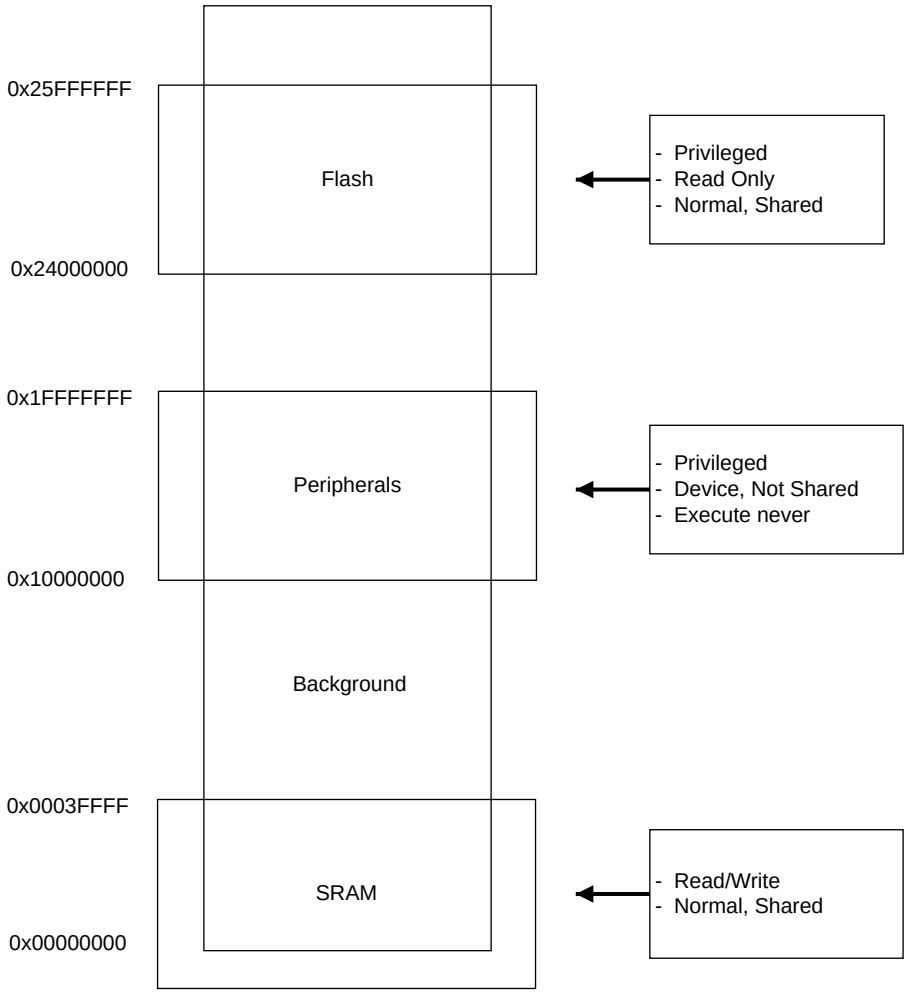
Figure 10-1: Single region and overlapping regions



For example, if an embedded system defines a large low priority region with privileged access only, it can then overlay a smaller region with user mode access permitted on top of this. The location of the smaller region can be moved over different areas of the privileged region on context switches to give a number of different user task-specific spaces. Each time that the smaller user-accessible region is moved, the previously covered area becomes protected by the privileged region. This means that with only one or two instructions writing to the appropriate CP15 register. you can provide an area of memory specific to each task, protected from all other tasks.

It is common practice to define a region that covers the entire 4GB physical address map, that has lower priority than any other region and that provides the memory attributes for accesses that do not match any of the other defined memory regions. If you require such accesses to generate a memory abort, this can be done using the SCTLR.BR bit, that provides this behavior without having to program region 0 to do so. Alternatively, the default memory map can be used to define the background region for privileged accesses.

Figure 10-2: Example Static MPU Region configuration with background region



The Base address of a region must always be aligned to the region size. This means, for example, that a 32KB region must have a base address aligned to a 32KB boundary.

Initialization and additional programming of the MPU regions is achieved through CP15 MCR instructions.

None of the regions are defined or enabled after reset. Any access that lies outside a defined and enabled region when the MPU is enabled will cause an abort. Therefore, at least one region must be defined before enabling the MPU after reset. If the MPU is enabled and no regions are defined, the processor enters a state from which it is recoverable only by an additional reset.

However, the Cortex-R series processors can be configured to permit privileged accesses to use the default memory map if no region has been defined for a particular address location.

The code enabling the MPU, by setting the SCTLR.M bit in the System Control Register, must be in a memory location that is defined as executable, otherwise the core will immediately take an abort exception when the MPU is enabled.

The available memory types are Normal, Device and Strongly Ordered. For Normal memory there are a number of possible cache policies, such as write-back and write-through, that can be selected for a region. This means that, for example, one region can be marked as using write-back cache policy, while another is noncacheable.

For the purposes of memory protection, it is the Access Control settings that are of interest. Access to a region in memory can be set as read-write, read-only, or no access. It is qualified by the current processor mode, that can be either privileged or non-privileged (user).

When the processor accesses a memory address, the MPU determines the attributes of the region applying to that address and compares the access permission attributes of the region with the current processor mode to determine what action is required. If the access is permitted by the region access criteria, the read or write to main memory (or TCM) occurs. If it is not permitted, the access to memory does not occur and an abort is generated. This is either a prefetch or data abort, and the appropriate abort handler is called.

The MPU in the Cortex-R series processors is software programmable. The protection configuration might change during a context switch and so should be reprogrammed.


Note

Usually there is no requirement to flush all caches whenever the MPU is reprogrammed. Doing so can significantly degrade system performance.

In the Cortex-R4 and Cortex-R5 processors, the presence of the programmable MPU is optional although it is generally included. If present, there might be either 8, 12 or 16 regions. The number of regions is defined by the hardware implementer at RTL configuration stage.

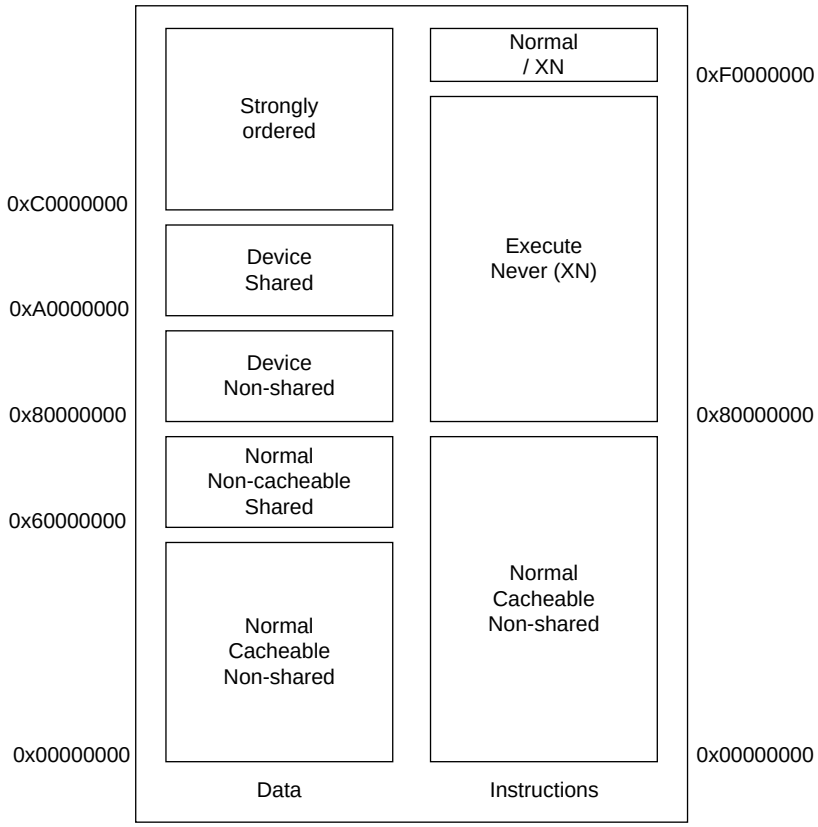
In the Cortex-R7 processor the programmable MPU is required and might have either 12 or 16 regions, as defined by the hardware implementer at RTL configuration stage.

The smallest size of a region in the Cortex-R4 and Cortex-R5 processor is 32 bytes. The smallest size of a region in the Cortex-R7 processor is 256 bytes.

If a region is of 256 bytes or more, it might be divided into 8 sub-regions.The regions are common to both instruction and data accesses. However, it is possible to use the Execute Never (XN) attribute to disallow instruction execution from a peripheral or data region.

If the MPU is disabled or not present, the processor uses the default memory map and default protection settings. Figure 10-3: Default memory map on page 110 shows the default memory map and the base address for each region.

Figure 10-3: Default memory map



Note:

- In the instruction memory type, the top 256MB is Normal Executable only when HIVECS is enabled. Otherwise it is treated as Execute Never.
- If instruction cache is disabled, then the Cacheable regions in the instruction memory type become Noncacheable.
- If data cache is disabled, then the Cacheable, Non-shared regions in the data memory type become Noncacheable, Shared.

10.2.1 Sub-Regions

Each region larger than 256 bytes can be split into eight equal sized non-overlapping sub-regions. This means the granularity of protection and memory attributes can be increased without significant increase in hardware complexity and reduces the number of regions that must be used to align protection boundaries to unaligned addresses.

Each of the sub-regions can be individually enabled or disabled. An access to a memory address in a disabled sub-region does not use the attributes and permissions defined for that region. Instead, it uses the attributes and permissions of a lower priority region or generates a background fault if no other regions overlap at that address. This enables increased protection and memory attribute granularity.

Disabling sub-regions provides backward compatibility with the ARMv6 architecture.

10.2.2 MPU memory region programming registers

The MPU memory region programming registers program the MPU regions.

There is one register that specifies which one of the sets of region registers is to be accessed. Each region has its own registers to specify:

- Region base address.
- Region size and enable.
- Region access control.

You can implement the processor with 12 or 16 regions, or entirely without an MPU. If you implement the processor without an MPU, then there are no regions and no region programming registers.

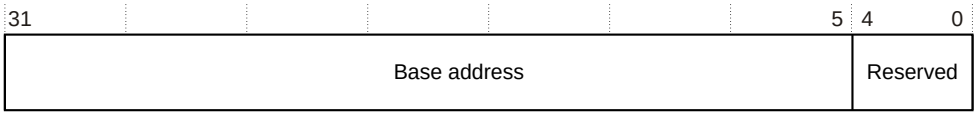
When the MPU is enabled:

- The MPU determines the access permissions for all accesses to memory, including the TCMs. Therefore, you must ensure that the memory regions in the MPU are programmed to cover the complete TCM address space with the appropriate access permissions. You must define at least one of the regions in the MPU.
- An access to an undefined area of memory normally generates a background fault.
- For the TCM space the processor uses the access permissions but ignores the region attributes from MPU.

c6, MPU Region Base Address Register

The MPU Region Base Address Register describes the base address of the region specified by the Memory Region Number Register.

Figure 10-4: Region base address register



To access an MPU Region Base Address Register, read or write CP15 with:

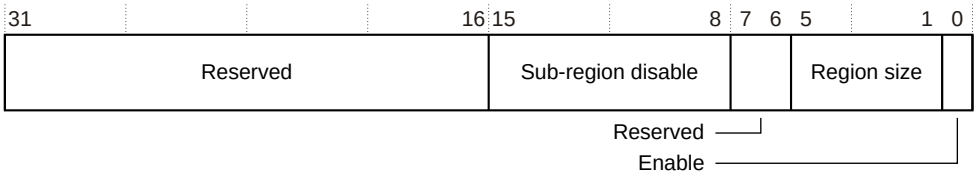
```
MRC p15, 0, <Rd>, c6, c1, 0 ; Read MPU Region Base Address Register
MCR p15, 0, <Rd>, c6, c1, 0 ; Write MPU Region Base Address Register
```

c6, MPU Region Size and Enable Register

The MPU Region Size and Enable Register specifies the size of the region specified by the Memory Region Number Register, identifies the address ranges that are used for a particular region and

enables or disables the region, and its sub-regions, specified by the Memory Region Number Register.

Figure 10-5: Region size and enable register



To access an MPU Region Size and Enable Register, read or write CP15 with:

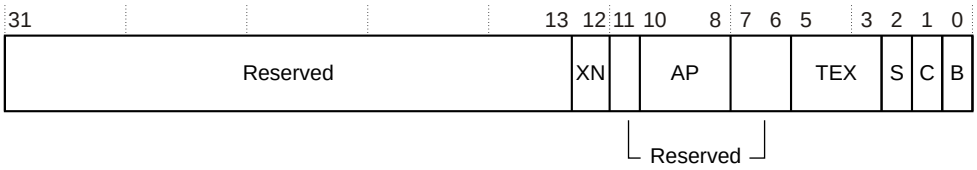
```
MRC p15, 0, <Rd>, c6, c1, 2 ; Read Data MPU Region Size and Enable Register
MCR p15, 0, <Rd>, c6, c1, 2 ; Write Data MPU Region Size and Enable Register
```

Writing a region size that is outside the range results in Unpredictable behavior.

c6, MPU Region Access Control Register

The MPU Region Access Control Register holds the region attributes and access permissions for the region specified by the Memory Region Number Register.

Figure 10-6: Region access control register



To access the MPU Region Access Control Registers read or write CP15 with:

```
MRC p15, 0, <Rd>, c6, c1, 4 ; Read MPU Region Access Control Register
MCR p15, 0, <Rd>, c6, c1, 4 ; Write MPU Region Access Control Register
```

To execute instructions in User and Privileged modes:

- The region must have read access as defined by the AP bits.
- The XN bit must be set to 0.

10.2.3 MPU control registers in CP15

This section provides a brief summary of the CP15 registers and instructions used to control the MPU.

- Memory Region Register Number (`MCR p15, 0, Rd, c6, c2, 0`)
  - Rd[3:0] contains the region number to select.

    - Controls which region number the following accesses apply to.
- Region Base (`MCR p15, 0, Rd, c6, c1, 0`)
    - Rd[31:5] contains MSBs of the base address.
    - This must be aligned to the size of the region. For example, if the size is 1KB, bits [9:5] must be zero.
- Region Size and Enable (`MCR p15, 0, Rd, c6, c1, 2`)
    - Rd[5:1] sets the size.
    - In the Cortex-R4 and Cortex-R5 processors, valid values are from `0b00100` (32 bytes) to `0b11111` (4GB).
    - In the Cortex-R7 processor, valid values are from `0b00111` (256 bytes) to `0b11111` (4GB).
    - Rd[0] is the enable bit.
    - At least one valid, enabled region is required prior to enabling the MPU.
- Region Access Control (`MCR p15, 0, Rd, c6, c1, 4`)
    - Rd[12]: Execute Never (XN) determines if a region of memory is Executable.
    - Rd[10:8]: Access Permission (AP).
    - Defines the data access permissions.
    - Rd[5:3]: Type Extensions (TEX) defines the Type Extensions attributes of the region.
    - Combined with the S, C and B bits to determine the memory type.
    - Rd[2]: Shared (S).
    - Rd[1]: Cacheable (C).
    - Rd[0]: Bufferable (B).

10.3 Memory attributes

The MPU specifies a number of attributes, including access permissions, memory type, and cache policies.

10.3.1 Memory Access Permissions

The Access Permission (AP) bits in the region configuration give the access permissions for a region. See Table 10-1: Summary of Access Permission encodings on page 114.

An access that does not have the necessary permission (or that faults) is aborted. On a data access, this will result in a precise data abort exception. On an instruction fetch, the access is marked as aborted and if the instruction is not subsequently flushed before execution, a prefetch abort exception is taken. Faults generated by an external access will not, in general, be synchronous.

Information about the address of the faulting location and the reason for the fault is stored in the fault address and fault status registers. The abort handler can then take appropriate action - for

example, modifying the region configuration to remedy the problem and then returning to the application to retry the access. If there is no available solution, the application that generated the abort must be terminated.

Table 10-1: Summary of Access Permission encodings

AP	Privileged	Unprivileged	Description
00	No access	No access	Permission fault
01	Read/Write	No access	Privileged Access only
10	Read/Write	Read	No user-mode write
11	Read/Write	Read/Write	Full access
00			Reserved
01	Read	No access	Privileged Read only
10	Read	Read	Read only
11			Reserved

10.3.2 Memory types

Earlier ARM architecture versions enabled you to specify the memory access behavior of regions by configuring whether the cache and write buffer could be used for that location. This simple scheme is inadequate for today's more complex systems and processors, where you can have multiple levels of caches, hardware managed coherency between multiple processors sharing memory and processors that can speculatively fetch both instructions and data.

Three mutually exclusive memory types are defined in the ARM architecture. All regions of memory are configured as one of these three types:

- Strongly-ordered
- Device
- Normal

These are used to describe the memory regions. A summary of the memory types is shown in Table 10-2: Memory attributes on page 114.

Table 10-2: Memory attributes

Memory type	Shareable/Non-shareable	Cacheable	Description
Normal	Shareable	Yes	Designed to handle normal memory that is shared between multiple cores.
Normal	Non-shareable	Yes	Designed to handle normal memory that is used only by a single core.
Device		No	Designed to handle memory-mapped peripherals. Shared memory was originally used to distinguish between accesses directed to the "peripheral private port" found on several ARM11 processors. All memory accesses to Device memory occur in program order.

Memory type	Shareable/Non-shareable	Cacheable	Description
Strongly-ordered		No	All memory accesses to Strongly-ordered memory occur in program order. All Strongly-ordered accesses are assumed to be shared.

Table 10-3: Memory type and cacheable properties encoding in the region configuration entry on page 115 shows how the TEX, C and B bits within the memory type and access control registers are used to set the memory types of a region and also the cache policies to be used. The meaning of each of the memory types is described in Chapter 10 Memory Ordering, while the cache policies were described in Chapter 7 Caches.

Table 10-3: Memory type and cacheable properties encoding in the region configuration entry

TEX	C	B	Description	Memory type
000	0	0	Strongly-ordered	Strongly-ordered
000	0	1	Shareable device	Device
000	1	0	Outer and Inner write-through, no allocate on write	Normal
000	1	1	Outer and Inner write-back, no allocate on write	Normal
001	0	0	Outer and Inner non-cacheable	Normal
001			Reserved	
010	0	0	Non-shareable device	Device
010			Reserved	
011			Reserved	
1XX	Y	Y	Cached memory XX = Outer policy YY = Inner policy	Normal

The final entry within the table requires more explanation. For normal cacheable memory, the two least significant bits of the TEX field are used to provide the outer cache policy, as in Table 10-4: Outer cache policy encoding on page 115, while the C and B bits give the inner cache policy (for level 1 and any other cache that is to be treated as inner cache). This enables you to specify different cache policies for both the inner and outer cache.

For the Cortex-R processors inner cache properties only apply to the L1 caches. On some older processors, outer cache might support write allocate, while the L1 cache might not. Such processors must still behave correctly when running code that requests this cache policy.

Table 10-4: Outer cache policy encoding

Memory attribute encoding	Cache policy
00	Non-cacheable
01	Write-back, write allocate
10	Write through, no write allocate
11	Write-back, no write allocate

10.3.3 Execute Never

When set, the Execute Never (XN) bit in the translation table entry prevents speculative instruction fetches taking place from required memory locations and will cause a prefetch abort to occur if execution from the memory location is attempted. Typically device memory regions are marked as Execute Never to prevent accidental execution from such locations, and to prevent undesirable side-effects that might be caused by speculative instruction fetches.

10.4 Attributes and cache maintenance

The MPU configuration is programmed using CP15. For each region you must specify:

- Base address.
- Size.
- Attributes.

The MPU can be programmed so that any given memory address can appear in more than one region, in which case the highest numbered matching region will set the attributes for that address. This is called 'resolving the attributes' for that address. The final attributes for the given address are termed the 'resolved attributes'.

MPU attributes for the Cortex-R series processors are:

- Memory type (Normal, Device, Strongly-Ordered).
- Cacheability.
- Shareability.
- Permission control (User/Privileged, Read/Write, Executable).


Note

The permission controls for access to a particular address are always controlled by the MPU. However the cacheability and shareability attributes for a region can be treated as hints by the processor and the attribute overriden.

For example, a TCM is by definition a Normal, Non-cacheable, Non-shareable memory region and so in the Cortex-R5 processor the TCM is always treated as Normal, Non-cacheable, Non-shareable memory, regardless of the MPU region settings.

Any of the above attributes can be changed without disabling and re-enabling the MPU, so long as care is taken to ensure that the resolved attributes for the code (and its data) performing the MPU update are not changed.

Cache maintenance is required when the memory type, cacheability, or shareability attributes for a memory location are changed by reprogramming the MPU. Coherency in both Level-1 Instruction

and Data caches and any Level-2 caches (if present) must then be restored by performing the following operations:

- I-cache: invalidate all.
- D-cache: clean and invalidate all.

For a description of how to do this for the Cortex-R7, see the examples in Boot Code.

It is not necessary to disable and re-enable the MPU while restoring coherency in this way.

An alternative strategy might be to invalidate by address (or clean and invalidate for the D-cache) for all addresses that used to be cacheable but now are not. However this will not usually result in improved performance.

Write-Through (WT) and Write-Back (WB) are both different cacheability attributes and any change to them will require cache maintenance, for example, changing from WB to WT is not safe without a clean and invalidate (or at least a clean).

10.5 Managing the MPU in context switches

Context switches that only change permission control attributes do not require cache maintenance operations. In this case, the caches continue to operate correctly without any loss of coherency.

In processors that use an MPU to control memory accesses, context switches will typically only modify the permission control for memory locations but will not modify the memory type, cacheability, or shareability attributes. Therefore context switching can be performed efficiently with the MPU remaining enabled and without having to do any cache maintenance.


Note

To enable or disable the MPU:

- The L1 caches must be invalidated.
- The L1 data cache must be cleaned.

However, when the MPU is reprogrammed, for example on a context switch, these steps are rarely necessary.

10.5.1 Permission modification in context switching

Suppose that the operating system programs the MPU as follows:

```
MPU Region 0:          Address from 0x00000000 to 0xFFFFFFFF
```

Described as Normal memory, Enabled, Cacheable using Write-Back

Executable = No. Permission = No Access

```
MPU Region 1:          Address from 0x10000000 to 0x1FFFFFFF
```

Described as Normal memory, Enabled, Cacheable using Write-Back

Executable = Yes. Permission = User Read/Write.

The address range `0x10000000` to `0x1FFFFFFF` corresponds to the memory range for Task 1 and part of Region 1 is cached during the task.

A context switch requires a move to Task 2 in which:

MPU Region 0: Stays the same

MPU Region 1: Is now relocated to address range `0x20000000` to `0x2FFFFFFF`, and therefore the addresses in the range `0x10000000` to `0x1FFFFFFF` hit in Region 0 and do not have access permission.

Because only the permissions have changed there is no requirement to perform cache maintenance when switching between Task 1 and Task 2.

If the processor attempts to read or write memory at `0x10001000` during Task 2, it will violate the access permission. In this case:

- It will generate a Synchronous Data Abort.
- The cache might be read, depending on the implementation, but the data will always be discarded.
- The target register will not be updated.
- If the address is not in the cache the processor will not start a line fill.
- Writes will not be issued to the memory system.


Note

When the MPU is configured and has switched from Task 1 to Task 2 as described above, any dirty cache lines corresponding to address range `0x10000000` to `0x1FFFFFFF` will still be evicted as normal, despite no longer being accessible for reads or writes.

10.6 Cache maintenance recommendations

It is sometimes necessary to perform cache maintenance operations, such as cleaning or invalidating the cache, when MPU region attributes are changed.

For the Cortex-R series processors, changing from a less restrictive to a more restrictive attribute requires cache maintenance. An example is when changing a region's resolved attribute from

cacheable to noncacheable. Therefore it is possible to identify changes in the memory type, cacheability, or shareability attributes that do not require cache maintenance operations.

However, ARM recommends that cache is always maintained when changes are made to the memory type, cacheability, or shareability attributes so that programs remain platform independent. Additional implications might also exist in the level-2 memory system that are outside the domain of the processor and are system-specific.

For any given memory location, ARM recommends that you have fixed values for the memory type, cacheability, or shareability attributes and that these attribute values are independent of the currently executing context.

Failure to guarantee this will mean that the OS must explicitly manage the mismatched attributes, that will involve cache maintenance and other considerations.


Note

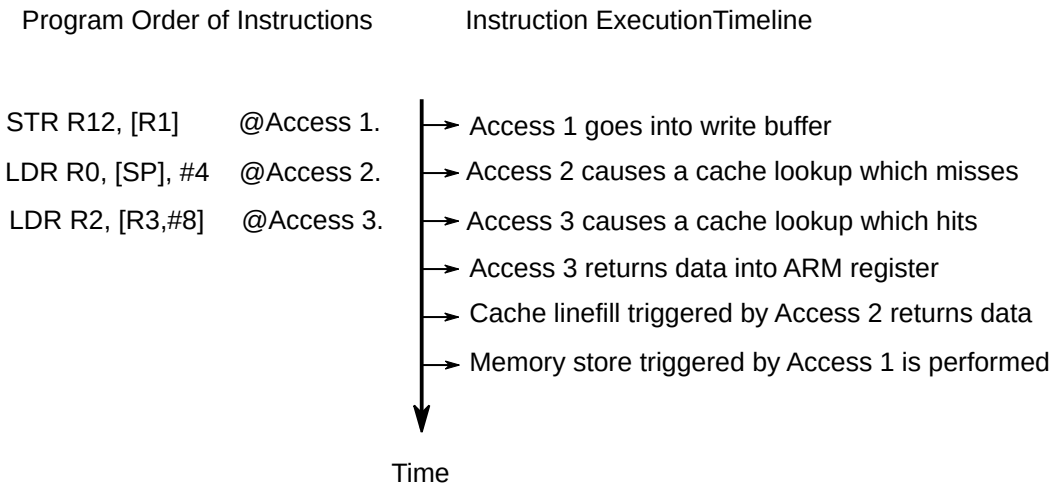
In most MPU based systems, attribute changes that require cache maintenance, such as changes to memory type or cacheability, do not typically occur after system start-up.

11. Memory Ordering

Early implementations of the ARM architecture such as the ARM7TDMI executed all instructions in program order. Each instruction was fully executed before the next instruction was started.

Newer processors employ a number of optimizations that relate to the order in which instructions are executed and the way memory accesses are performed. As we have seen, the speed of execution of instructions by the core is significantly higher than the speed of external memory. Caches and write buffers are used to partially hide the latency associated with this difference in speed. One potential effect of this is for memory addresses to be re-ordered. This would mean that the order in which load and store instructions are executed by the core will not necessarily be the same as the order in which the accesses are seen by external devices.

Figure 11-1: Memory ordering example



In Figure 11-1: Memory ordering example on page 120, we have three instructions listed in program order. The first instruction performs a write to external memory that in this example, misses in the cache (Access 1). It is followed in program order by two reads, one that misses in the cache (Access 2) and one that hits in the cache (Access 3). Both of the read accesses could complete before the write buffer completes the write associated with Access 1. Hit-under-miss behaviors in the cache mean that a load that hits in the cache (like Access 3) can complete before a load earlier in the program that missed in the cache (like Access 2).

It is still possible to preserve the illusion that the hardware executes instructions in the order you wrote them. There are generally only a few cases where you have to worry about such effects. For example, if you are modifying CP15 registers, copying or otherwise changing code in memory, it might be necessary to explicitly make the core wait for such operations to complete.

For very high performance cores that support speculative data accesses, multi-issuing of instructions, cache coherency protocols and out-of-order execution to make additional performance gains, there are even greater possibilities for re-ordering. In general, the effects of this re-ordering are invisible to you, in a single core system. The hardware takes care of many

possible hazards. It will ensure that data dependencies are respected and ensure the correct value is returned by a read, allowing for potential modifications caused by earlier writes.

However, in cases where you have multiple cores that communicate through shared memory (or share data in other ways), memory ordering considerations become more important. In general, you are most likely to care about exact memory ordering at points where multiple execution threads must be synchronized.

Processors that conform to the ARM v7-R architecture employ a weakly-ordered model of memory, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The model can reorder memory read operations (from `LDR`, `LDM` and `LDD` instructions) with respect to each other, to store operations, and certain other instructions. Reads and writes to Normal memory can be re-ordered by hardware, with such re-ordering being subject only to data dependencies and explicit memory barrier instructions. In cases where stronger ordering rules are observed, this is communicated to the processor through the memory type attribute of the region configuration that describes that memory. Enforcing ordering rules on the core limits the possible hardware optimizations and therefore reduces performance and increases power consumption.

11.1 ARM memory ordering model

As we have said, Cortex-R series processors employ a weakly ordered memory model. However, within this model specific regions of memory can be marked as Strongly-ordered. In this case memory transactions are guaranteed to occur in the order they are issued,

Three mutually exclusive memory types are defined. All regions of memory are configured as one of these three types:

- Strongly-ordered
- Device
- Normal

In addition, for Normal and Device memory, it is possible to specify whether the memory is Shareable (accessed by other agents) or not. For Normal memory, Inner and Outer cacheable properties can be specified.

In Table 11-1: Memory type access order on page 121 A1 and A2 are two accesses to non-overlapping addresses. A1 occurs before A2 in program code, but writes can be issued out of order.

Table 11-1: Memory type access order

	A2: Normal	A2: Device	A2: Strongly-ordered
A1: Normal	No order enforced	No order enforced	No order enforced
A1: Device	No order enforced	Issued in program order	Issued in program order
A1: Strongly-ordered	No order enforced	Issued in program order	Issued in program order

11.1.1 Strongly-ordered and Device memory

Accesses to Strongly-ordered and Device memory have the same memory-ordering model. Access rules for this memory are as follows:

- The number and size of accesses is preserved. Accesses are atomic, and will not be interrupted part way through.
- Both read and write accesses can have side effects. Accesses are never cached. Speculative accesses will never be performed.
- Accesses cannot be unaligned.
- The order of accesses arriving at Device memory is guaranteed to correspond to the program order of instructions that access Strongly-ordered or Device memory. This guarantee applies only to accesses within the same peripheral or block of memory. The size of such a block is implementation defined, but has a minimum size of 1KB.
- In the ARMv7 architecture, the processor can re-order Normal memory accesses around Strongly-ordered or Device memory accesses.

The only difference between Device and Strongly-ordered memory is that:

- A write to Strongly-ordered memory can complete only when it reaches the peripheral or memory component accessed by the write.
- A write to Device memory is permitted to complete before it reaches the peripheral or memory component accessed by the write.

System peripherals will almost always be mapped as Device memory.

Regions of Device memory type can be described using the Shareable attribute.

On some ARMv6 processors, the Shareable attribute of Device accesses is used to determine which memory interface is used for the access, with memory accesses to areas marked as Device, Non-Shareable performed using a dedicated interface, the private peripheral port. This mechanism is not used on ARMv7 processors.


Note

These memory ordering rules provide guarantees only about explicit memory accesses (those caused by load and store instructions). The architecture does not provide similar guarantees about the ordering of instruction fetches with respect to such explicit memory accesses.

11.1.2 Normal memory

Normal memory is used to describe most parts of the memory system. All ROM and RAM devices are considered to be Normal memory. All code to be executed by the processor must be in Normal

memory. The architecture does not permit code to be in a region of memory that is marked as Device or Strongly-ordered.

The properties of Normal memory are as follows:

- The processor can repeat read and some write accesses.
- The processor can pre-fetch or speculatively access additional memory locations, with no side effects (if permitted by MPU access permission settings). The processor will not perform speculative writes, however.
- Unaligned accesses can be performed.
- Multiple accesses can be merged by processor hardware into a smaller number of accesses of a larger size. Multiple byte writes could be merged into a single double-word write, for example.

Cacheability attributes

Regions of Normal memory must also have cacheability attributes described (see Caches for details of the supported cache policies). The ARM architecture supports cacheability attributes for Normal memory for two levels of cache, the inner and outer cache. The mapping between these levels of cache and the implemented physical levels of cache is implementation defined.

Inner refers to the innermost caches, and always includes the processor Level 1 cache. An implementation might not have any outer cache, or it can apply the outer cacheability attribute to an Level 2 cache. For example, in a system containing a Cortex-R7 processor and the L2C-310 Level 2 cache controller, the L2C-310 is considered to be the outer cache.


Note

Some parts of memory such as those containing peripheral devices are non-cacheable.

Shareable attributes

Normal memory must also be identified either as Shareable or Non-Shareable. A region of Normal memory with the Non-Shareable attribute is one that is used only by this core. There is no requirement for the core to make accesses to this location coherent with other cores. If other cores do share this memory, any coherency issues must be handled in software. For example, this can be done by having individual cores perform cache maintenance and barrier operations.

A region with the Shareable attribute set is one that can be accessed by other agents in the system. Accesses to memory in this region by other processors within the same shareability domain are coherent. This means that you can safely ignore the effects of data or caches. Without the Shareable attribute, in situations where cache coherency is not maintained between processors for a region of shared memory, you would have to explicitly manage coherency yourself.

The ARMv7 architecture enables you to specify Shareable memory as Inner Shareable or Outer Shareable (this latter case means that the location is both Inner and Outer Shareable). The Cortex-R processors do not distinguish between outer and inner shareable memory.

11.2 Memory barriers

A memory barrier is an instruction that requires the processor to apply an ordering constraint between memory operations that occur before and after the memory barrier instruction in the program. Such instructions are also known as memory fences in other architectures.

The term memory barrier can also be used to refer to a compiler mechanism that prevents the compiler from scheduling data access instructions across the barrier when performing optimizations. For example in GCC, you can use the inline assembler memory clobber, to indicate that the instruction changes memory and therefore the optimizer cannot re-order memory accesses across the barrier. The syntax is as follows:

```
asm volatile("" ::: "memory");
```

ARM RealView Compilation Tools (RVCT) includes a similar intrinsic, called `__schedule_barrier()`.

Here, however, we are looking at hardware memory barriers, provided through dedicated ARM assembly language instructions. As we have seen, processor optimizations such as caches, write buffers and out-of-order execution can result in memory operations occurring in an order different from that specified in the executing code. Normally, this re-ordering is invisible to you. Application developers do not normally have to worry about memory barriers. However, there are cases where you might have to take care of such ordering issues, for example in device drivers or when you have multiple observers of the data that have to be synchronized.

The ARM architecture specifies memory barrier instructions, that enable you to force the core to wait for memory accesses to complete. These instructions are available in both ARM and Thumb code, in both user and privileged modes. In older versions of the architecture, these were performed using CP15 operations in ARM code only. Use of these is now deprecated, although preserved for compatibility.

Let's start by looking at the practical effect of these instructions in a single core processor. This description is a simplified version of that given in the ARM Architecture Reference Manual, what is written here is intended to introduce the usage of these instructions. The term explicit access is used to describe a data access resulting from a load or store instruction in the program. It does not include instruction fetches.

Data Synchronization Barrier (DSB)

This instruction forces the processor to wait for all pending explicit data accesses to complete before any additional instructions stages can be executed. There is no effect on pre-fetching of instructions.

Data Memory Barrier (DMB)

This instruction ensures that all memory accesses in program order before the barrier are observed in the system before any explicit memory accesses that appear in program order after the barrier. It does not affect the ordering of any other instructions executing on the processor, or of instruction fetches.

Instruction Synchronization Barrier (ISB)

This flushes the pipeline and prefetch buffer(s) in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has completed. This ensures that the effects of context altering operations (for example, CP15 or branch predictor operations), executed before the ISB instruction are visible to any instructions fetched after the ISB. This does not in itself cause synchronization between data and instruction caches, but is required as a part of such an operation.

Several options can be specified with the DMB or DSB instructions, to provide the type of access and the shareability domain it should apply to, as follows:

SY

This is the default and means that the barrier applies to the full system, including all processors and peripherals.

ST

A barrier that waits only for stores to complete.

ISH

A barrier that applies only to the Inner Shareable domain.

ISHST

A barrier that combines ST and ISH (that is, it only stores to the Inner Shareable).

NSH

A barrier only to the Point of Unification (PoU). (See Point of coherency and unification).

NSHST

A barrier that waits only for stores to complete and only out to the point of unification.

OSH

Barrier operation only to the Outer Shareable domain.

OSHST

Barrier operation that waits only for stores to complete, and only to the Outer Shareable domain.

To make sense of this, you must use a more general definition of the DMB and DSB operations in a multi-core system. The use of the word processor (or agent) in the following text does not necessarily mean a processor and also could refer to a DSP, DMA controller, hardware accelerator or any other block that accesses shared memory.

The DMB instruction has the effect of enforcing memory access ordering within a shareability domain. All processors within the shareability domain are guaranteed to observe all explicit memory accesses before the DMB instruction, before they observe any of the explicit memory accesses after it.

The DSB instruction has the same effect as the DMB, but in addition to this, it also synchronizes the memory accesses with the full instruction stream, not just other memory accesses. This means that when a DSB is issued, execution will stall until all outstanding explicit memory accesses have completed. When all outstanding reads have completed and the write buffer is drained, execution resumes as normal.

It might be easier to appreciate the effect of the barriers by considering an example. Consider the case of a dual-core Cortex-R7 processor. These cores operate as an SMP cluster and form a single Inner Shareable domain. When a single core within the cluster executes a `DMB` instruction, that core will ensure that all data memory accesses in program order before the barrier complete, before any explicit memory accesses that appear in program-order after the barrier. This way, it can be guaranteed that all cores within the cluster will see the accesses on either side of that barrier in the same order as the core that performs them.

11.2.1 Memory barrier use example

Consider the case where you have two cores A and B and two addresses in Normal memory (Addr1 and Addr2) held in core registers. Each core executes two instructions as follows:

Core A:

```
STR R0, [Addr1]
LDR R1, [Addr2]
```

Core B:

```
STR R2, [Addr2]
LDR R3, [Addr1]
```

Here, there is no ordering requirement and you can make no statement about the order in which any of the transactions occur. The addresses `Addr1` and `Addr2` are independent and there is no requirement on either core to execute the load and store in the order written in the program, or to care about the activity of the other core.

There are therefore four possible legal outcomes of this piece of code, with four different sets of values from memory ending up in core A, register R1 and core B, register R3:

- A gets the old value, B gets the old value.
- A gets the old value, B gets the new value.
- A gets the new value, B gets the old value.
- A gets the new value, B gets the new value.

If it were possible to involve a third core, C, you must also note that there is no requirement that it would observe either of the stores in the same order as either of the other cores. It is perfectly permissible for both A and B to see an old value in `Addr1` and `Addr2`, but for C to see the new values.

Consider the case where the code on B looks for a flag being set by A and then reads memory, for example, if you are passing a message from A to B. You might have code similar to that below:

Core A:

```
STR R0, [Msg]       @ write some new data into postbox
```

```
STR R1, [Flag]      @ new data is ready to read
```

Core B:

```
Poll_loop:
 LDR R1, [Flag]
 CMP R1,#0     @ is the flag set yet?
 BEQ Poll_loop
 LDR R0, [Msg] @ read new data.
```

Again, this might not behave in the way that is expected. There is no reason why core B is not permitted to speculatively perform the read from `[Msg]` before the read from `[Flag]`. This is normal, weakly-ordered memory and the core has no knowledge of a possible dependency between the two. You must explicitly enforce the dependency by inserting a memory barrier. In this example, you actually require two memory barriers. Core A requires a `DMB` between the two store operations, to make sure they happen in the order you originally specified. Core B requires a `DMB` before the `LDR R0, [Msg]` to be sure that the message is not read until the flag is set.

11.2.2 Avoiding deadlocks with a barrier

Another situation that can cause a deadlock if barrier instructions are not used is where a core writes to an address and then polls for an acknowledge value to be applied by a peripheral.

The following example shows the type of code that can cause a problem.

```
STR R0, [Addr] @ write a command to a peripheral register
DSB
Poll_loop:
 LDR R1, [Flag]
 CMP R1,#0 @ wait for an acknowledge/state flag to be set
 BEQ Poll_loop
```

The ARMv7 architecture without multiprocessing extensions does not strictly require the core's store to `[Addr]` to ever complete (it could be sitting in a write buffer while the memory system is kept busy reading the flag), so both cores could potentially deadlock, each waiting for the other. Inserting a `DSB` after the `STR` of the core forces its store to be observed before it will read from `Flag`.

11.2.3 WFE and WFI Interaction with barriers

The `WFE` (Wait For Event) and `WFI` (Wait For Interrupt) instructions enable you to stop execution and enter a low-power state. To ensure that all memory accesses prior to executing `WFI` or `WFE` have been completed (and made visible to other cores), you must insert a `DSB` instruction.

An additional consideration relates to usage of `WFE` and `SEV` (Send Event) in an MP system. These instructions enable you to reduce the power consumption associated with a lock acquire loop (a spinlock). A processor that is attempting to acquire a mutex can find that some other processor

already has the lock. Instead of having the processor repeatedly poll the lock, you can suspend execution and enter a low-power state, using the WFE instruction.

The core wakes either when an interrupt or other asynchronous exception is recognized, or another core sends an event (with the SEV instruction). The core that had the lock will use the SEV instruction to wake-up other cores in the WFE state after the lock has been released. For the purposes of memory barrier instructions, the event signal is not treated as an explicit memory access. You therefore have to take care that the update to memory that releases the lock is actually visible to other processors before the SEV instruction is executed. This requires the use of a DSB. DMB is not sufficient as it only affects the ordering of memory accesses without synchronizing them to a particular instruction, whereas DSB will prevent the SEV from executing until all preceding memory accesses have been seen by other cores.

11.3 Cache coherency implications

The caches are largely invisible to the application programmer. However they can become visible when memory locations are changed elsewhere in the system or when memory updates made from the application code must be made visible to other parts of the system.

A system containing an external DMA device and a core provides a simple example of possible problems. There are two situations in which a breakdown of coherency can occur. If the DMA reads data from main memory while newer data is held in the core cache, the DMA will read the old data. Similarly, if a DMA writes data to main memory and stale data is present in the core cache, the core can continue to use the old data.

Dirty data in the core data cache must be explicitly cleaned before the DMA starts. Similarly, if the DMA is copying data to be read by the core, it must be certain that the core data cache does not contain stale data. The cache will not be updated by the DMA writing memory and this might require the core to clean or invalidate the affected memory areas from the caches before starting the DMA. As all ARMv7-R processors can do speculative memory accesses, it will also be necessary to invalidate after using the DMA.

11.3.1 Issues with copying code

Boot code, kernel code or JIT compilers can copy programs from one location to another, or modify code in memory. There is no hardware mechanism to maintain coherency between instruction and data caches. You must invalidate stale code from the instruction cache by invalidating the affected areas, and ensure that the code written has actually reached the main memory. Specific code sequences including instruction barriers are required if the core is then intended to branch to the modified code.

11.3.2 Compiler re-ordering optimizations

It is important to understand that memory barrier instructions apply only to hardware re-ordering of memory accesses. Inserting a hardware memory barrier instruction might not have any direct

effect on compiler re-ordering of operations. The `volatile` type qualifier in C tells the compiler that the variable can be changed by something other than the currently executing code that is accessing it. This is often used for C language access to memory mapped I/O, enabling such devices to be safely accessed through a pointer to a `volatile` variable. The C standard does not provide rules relating to the use of `volatile` in systems with multiple cores. So, although you can be sure that `volatile` loads and stores will happen in program specified order with respect to each other, there are no such guarantees about re-ordering of accesses relative to non-volatile loads or stores. This means that `volatile` does not provide a shortcut to implement mutexes.

12. Exceptions and Interrupts

An exception is any condition that requires the core to halt normal execution and instead execute a dedicated software routine known as an exception handler. There is typically an exception handler associated with each exception type. Exceptions are conditions or system events that usually requires remedial action or an update of system status by privileged software to ensure smooth functioning of the system. This is called handling an exception.

When the exception has been handled, privileged software prepares the core to resume whatever it was doing before taking the exception. Other architectures might refer to what ARM calls exceptions as traps or interrupts, however, in the ARM architecture, these terms are reserved for specific types of exceptions, described in Types of exception.

In normal program execution, the program counter increments through the address space, with explicit branches in the program modifying the flow of execution, for example, for function calls, loops, and conditional code. When an exception occurs, this pre-determined sequence of execution is interrupted, and temporarily switches to a routine to handle the exception.

In addition to responding to external interrupts, there are a number of other things that can cause the core to take an exception, both external, such as resets, external aborts from the memory system, and internal, such calls using the SVC instruction. You will recall from ARM Processor modes and Registers that dealing with exceptions causes the core to switch between modes and copy some registers into others. Readers new to the ARM architecture might want to refresh their understanding of the modes and registers described in ARM Processor modes and Registers, before continuing with this chapter.

12.1 Types of exception

ARM Processor modes and Registers described how the ARMv7-R architecture supports a number of processor modes, six privileged modes called FIQ, IRQ, Supervisor, Abort, Undefined and System, and the non-privileged User mode. The current mode can change under privileged software control or automatically when taking an exception.

Unprivileged user mode cannot directly affect the exception behavior of a core, but can generate an SVC exception to request privileged services. This is how user applications requests the Operating System to accomplish tasks on behalf of them. The unprivileged User mode can switch to another mode only by generating an exception.

When an exception occurs, the core saves the current status and the return address, enters a specific mode and possibly disables hardware interrupts. Execution handling for a given exception starts from a fixed memory address called an exception vector for that exception. Privileged software can program the location of a set of exception vectors into system registers, and they are executed automatically when respective exceptions are taken.

The following types of exception exist:

Interrupts

Processors that implement the ARMv7-R profile provide two interrupt types, called IRQ and FIQ.

FIQ is higher priority than IRQ. FIQ also has some potential speed advantages owing to its position in the vector table and the higher number of banked registers available in FIQ mode. This potentially saves clock cycles on pushing registers to the stack within the handler. Both of these kinds of exception are typically associated with input pins on the processor. External hardware asserts an interrupt request line and the corresponding exception type is raised when the current instruction finishes executing, assuming that the interrupt is not disabled.

Aborts

Aborts can be generated either on failed instruction fetches (prefetch aborts) or failed data accesses (data aborts). They can come from the external memory system giving an error response on a memory access. This might indicate that the specified address does not correspond to real memory in the system. Alternatively, the abort can be generated by the MPU.

An instruction can be marked within the pipeline as aborted, when it is fetched. The prefetch abort exception is taken only if the core then tries to execute it. The exception takes place before the instruction executes. If the pipeline is flushed before the aborted instruction reaches the execute stage of the pipeline, the abort exception will not occur. A data abort exception happens as a result of a load or store instruction and is considered to happen after the data read or write has been attempted.

An abort is described as synchronous if it is generated as a result of execution or attempted execution of the instruction stream, and where the return address will provide details of the instruction that caused it.

An asynchronous abort is not generated by executing instructions, while the return address might not always provide details of what caused the abort.

The ARMv7 architecture distinguishes between precise and imprecise asynchronous aborts. Aborts generated by the MPU are always synchronous. The architecture does not require particular classes of externally aborted accesses to be synchronous.

For precise asynchronous aborts, the abort handler can be certain which instruction caused the abort and that no additional instructions were executed after that instruction. This is in contrast to an imprecise asynchronous abort, the result when the external memory system reports an error on an unidentifiable access.

In this case, the abort handler cannot determine which instruction caused the problem, or if additional instructions might have executed after the one that generated the abort.

For example, if a buffered write receives an error response from the external memory system, additional instructions will have been executed after the store. This means that it is impossible for the abort handler to fix the problem and return to the application. All it can do is to kill the application that caused the problem. Device probing therefore requires special handling, as

externally reported aborts on reads to non-existent areas will generate imprecise synchronous aborts even when such memory is marked as Strongly-ordered, or Device.

Detection of asynchronous aborts is controlled by the CPSR A bit. If the A bit is set, asynchronous aborts from the external memory system are recognized by the core, but no abort exception is generated. Instead, the core keeps the abort pending until the A bit is cleared and takes an exception at that time. Kernel code will use a barrier instruction to ensure that pending asynchronous aborts are recognized against the correct application. If a thread has to be killed because of an imprecise abort, it must be the correct one.

Reset

All cores have a reset input and will take the reset exception immediately after they have been reset. It is the highest priority exception and cannot be masked. This exception is used to execute code on the core to initialize it, after power up

Exception generating instructions There are two classes of instruction that can cause exceptions on an ARM core. The first is the Supervisor Call (svc). This is typically used to provide a mechanism by which User mode programs can pass control to privileged, kernel code in the OS to perform OS-level tasks. The second is an undefined instruction. The architecture defines certain bit-patterns as corresponding to undefined opcodes. Trying to execute one of these causes an undefined Instruction exception to be taken. Executing coprocessor instructions for which there is no corresponding coprocessor hardware will also cause this trap to happen. Some instructions can be executed only in a privileged mode and executing these from User mode will cause an undefined instruction exception.

When an exception occurs, the core executes the handler corresponding to that exception. The location in memory where the handler is stored is called the exception vector. In the ARM architecture, exception vectors are stored in a table, called the exception vector table. Vectors for individual exception are located at fixed offsets from the beginning of the table. The table base is programmed in a system register by privileged software so that the core can locate the respective handler when an exception occurs. The fixed offsets for exceptions are shown in Table 12-1: Summary of exception behavior on page 133.

You can write the exception handlers in either ARM or Thumb code. The CP15 SCTLR.TE bit specifies whether exception handlers use ARM or Thumb code. When handling exceptions, the prior mode, state, and registers of the processor must be preserved so that the program can be resumed after the exception has been handled.

12.2 Exception priorities

When exceptions occur simultaneously, each exception is handled in turn before returning to the original application. It is not possible for all exceptions to occur concurrently. For example, the Undefined instruction (Undef) and supervisor call (SVC) exceptions are mutually exclusive because they are both triggered by executing an instruction.

Note The ARM architecture does not define when asynchronous exceptions are taken. Therefore the prioritization of asynchronous exceptions relative to other exceptions, both synchronous and asynchronous, is implementation defined.

All exceptions disable IRQ on entry to the handler, only FIQ and reset disable FIQ. This is done by the core automatically setting the CPSR I (IRQ) and F (FIQ) bits.

So, unless the handler explicitly disables it, an FIQ exception can interrupt an abort handler or IRQ exception. In the case of a data abort and FIQ occurring simultaneously, the data abort is taken first. This lets the core record the return address for the data abort. But as FIQ is not disabled by data abort, the core then takes the FIQ exception immediately. At the end of the FIQ you return back to the data abort handler.

Exception handling on the core is controlled through the use of an area of memory called the vector table. This is located by default at the bottom of the memory map in word-aligned addresses from `0x00` to `0x1C`. Most of the cached cores enable the vector table to be moved from `0x0` to `0xFFFF0000`.

Table 12-1: Summary of exception behavior

Normal Vector offset	High vector address	Exception
`0x0`	`0xFFFF0000`	Not used
`0x4`	`0xFFFF0004`	undefined instruction
`0x8`	`0xFFFF0008`	Supervisor Call
`0xC`	`0xFFFF000C`	Prefetch Abort
`0x10`	`0xFFFF0010`	Data Abort
`0x14`	`0xFFFF0014`	Not used
`0x18`	`0xFFFF0018`	IRQ interrupt
`0x1C`	`0xFFFF001C`	FIQ interrupt

12.2.1 Exception mode summary

Table 12-2: CPSR behavior on page 133 lists the state of the interrupt disabling I and F bits of the CPSR on entering an exception handler.

Table 12-2: CPSR behavior

Exception	Mode	CPSR interrupt mask
Reset	Supervisor	F = 1 I = 1
undefined instruction	Undef	I = 1
Supervisor Call	Supervisor	I = 1
Prefetch Abort	Abort	I = 1
Data Abort	Abort	I = 1
Not used		
IRQ interrupt	IRQ	I = 1
FIQ interrupt	FIQ	F = 1 I = 1

12.2.2 The Vector table

The first column in Table 12-2: CPSR behavior on page 133 gives the vector offset within the vector table associated with the particular type of exception. This is a table of instructions that the ARM processor jumps to when an exception is raised. These instructions are located in a specific place in memory. The normal vector base address is `0x00000000`, but Cortex-R Series processors enable the vector base address to be moved to `0xFFFF0000` (or `HIVECS`).

You can discover whether HIVECS is in use by reading bit [13], the SCTLR.V bit using similar code to the example in System Control Register (SCTLR).

You will notice that there is a single word address associated with each exception type. Therefore, only a single instruction can be placed in the vector table for each exception (although, in theory, two 16-bit Thumb instructions could be used). FIQ is different. See, FIQ and IRQ. Therefore, the vector table entry almost always contains one of the various forms of branches.

- `B <label>`

  This performs a PC-relative branch. It is suitable for calling exception handler code that is close enough in memory that the 24-bit field provided in the branch instruction is large enough to encode the offset.

- `LDR PC, [PC, #offset]`

  This loads the PC from a memory location whose address is defined relative to the address of the exception instruction. This lets the exception handler be placed at any arbitrary address within the full 32-bit memory space, but takes some extra cycles relative to the simple branch.

12.2.3 FIQ and IRQ

FIQ is typically reserved for a single, high-priority interrupt source that requires a guaranteed fast response time, with IRQ used for all of the other interrupts in the system.

As FIQ is the last entry in the vector table, the FIQ handler can be placed directly at the vector location and run sequentially from that address. This avoids a branch instruction and any associated delay, speeding up FIQ response times. The extra banked registers available in FIQ mode relative to other modes enables state to be retained between calls to the FIQ handler, again increasing execution speed by removing the need to push some registers before using them.

A additional key difference between IRQ and FIQ is that the FIQ handler is not expected to generate any other exceptions. FIQ is therefore reserved for special system-specific devices that have all their memory mapped and no need to make SVC calls to access kernel functions (so FIQ can be used only by code that does not have to use the kernel API).

12.2.4 The return instruction

The Link Register (LR) is used to store the appropriate return address for the PC after the exception has been handled. Its value has to be modified as shown in Table 12-3: Link Register Adjustments on page 135, depending on the type of exception occurred. The ARM Architecture Reference Manual defines the LR values that are appropriate (the definition derives from the values that were convenient for early hardware implementations).

Table 12-3: Link Register Adjustments

Exception	Adjustment	Return instruction	Instruction returned to
SVC	0	`MOVS PC, R14`	Next instruction
Undef	0	`MOVS PC, R14`	Next instruction
Prefetch Abort	-4	`SUBS PC, R14, #4`	Aborting instruction
Data abort	-8	`SUBS PC, R14, #8`	Aborting instruction if precise
FIQ	-4	`SUBS PC, R14, #4`	Next instruction
IRQ	-4	`SUBS PC, R14, #4`	Next instruction

12.3 Exception handling

When an exception occurs, the ARM core automatically does the following:

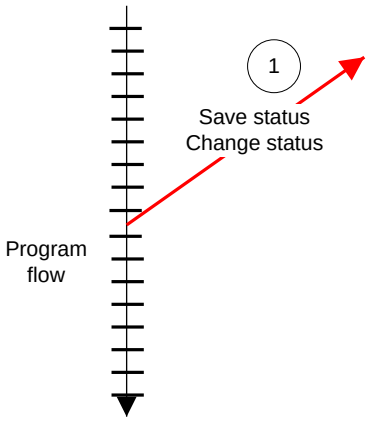
1. Copies the CPSR to the `SPSR_<mode>`, the banked register specific to the (non-user) mode of operation.
2. Stores a return address in the Link Register (LR) of the new mode.
3. Modifies the CPSR mode bits to the mode associated with the exception type.

- The other CPSR mode bits are set to values determined by bits in the CP15 System Control Register.
- The T bit is set to the value given by the CP15 TE bit.
- The J bit is cleared and the E bit (Endianness) is set to the value of the EE (Exception Endianness) bit.

This enables exceptions to always run in ARM or Thumb state and in little or big-endian, irrespective of the state the core was in before the exception.

1. Sets the PC to point to the relevant instruction from the exception vector table.

Figure 12-1: Taking the exception



When in the new mode the core will access the register associated with that mode, as shown in Figure 12-1: Taking the exception on page 136.

It will almost always be necessary for the exception handler software to save registers onto the stack immediately on exception entry. FIQ mode has more banked registers and so a simple handler might be able to be written in a way that requires no stack usage.

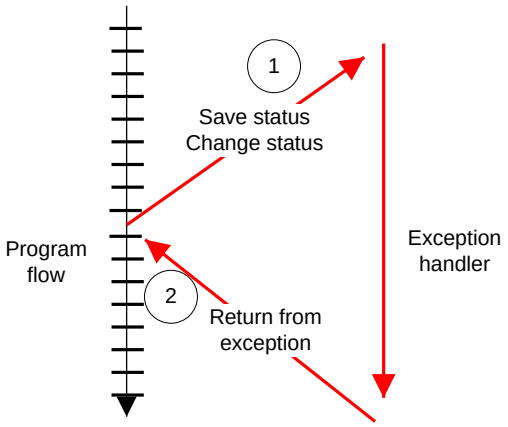
A special assembly language instruction is provided to assist with saving the necessary registers, called SRS (Store Return State). This instruction pushes the LR and SPSR onto the stack of any mode; the stack to be used is specified by the instruction operand.

12.3.1 Exit from an exception handler

To return from an exception handler, two separate operations must take place atomically:

1. Restore the CPSR from the saved SPSR.
2. Set the PC to the return address, see Figure 12-2: Returning from an exception on page 137.

Figure 12-2: Returning from an exception



In the ARM architecture this can be achieved either by using the `RFE` instruction or any flag-setting data processing operation (with the `s` suffix) with the PC as the destination register, such as `SUBS PC, LR, #offset` (note the `S`). The Return From Exception (`RFE`) instruction pops the link register and SPSR off the current mode stack.

There are a number of ways to achieve this.

- You can use a data processing instruction to adjust and copy the LR into the PC, for example:

  ```
  SUBS pc, lr, #4
  ```

  Specifying the `s` means the SPSR is copied to the CPSR at the same time.

  If the exception handler entry code uses the stack to store registers that must be preserved while it handles the exception, it can return using a load multiple instruction with the ^ qualifier. For example, an exception handler can return in one instruction using:

  ```
  LDMFD sp! {pc}^
  LDMFD sp!,{R0-R12,pc}^
  ```

  The `^` qualifier in this example means the SPSR is copied to the CPSR at the same time.

  To do this, the exception handler must save the following onto the stack:

  - All the work registers in use when the handler is invoked.
  - The link register, modified to produce the same effect as the data processing instructions.


Note

You cannot use 16-bit Thumb instructions to return from exceptions because these are unable to restore the CPSR.

- The `RFE` instruction restores the PC and SPSR from the stack of the current mode.

```
RFEFD sp!
```

12.4 Other exception handlers

This section briefly describes handlers for aborts, undefined instructions and svc instructions and considers how interrupts are handled by the Linux kernel. Reset handlers are covered in depth in Boot Code.

12.4.1 Abort handler

Abort handler code can vary significantly between systems. In many embedded systems, an abort indicates an unexpected error and the handler will record any diagnostic information, report the error and have the application (or system) quit gracefully.

CP15 registers provide the address of the memory access that caused the abort (the Fault Address Register) and the reason for the abort (Fault Status Register). The reason might be lack of access permission or an external abort. In addition, the link register (with a -8 or -4 adjustment, depending on whether the abort was caused by an instruction fetch or a data access), gives the address of the instruction executing before the abort exception. By examining these registers, the last instruction executed and possibly other things in the system (for example, page table entries), the abort handler can determine what action to take.

12.4.2 Undefined instruction handling

An undefined instruction exception is taken if the processor tries to execute an instruction with an opcode that is described as undefined in the ARM architecture specification, or when a coprocessor instruction is executed but no coprocessor recognizes it as an instruction that it can execute.

In some systems, it is possible that code includes instructions for a coprocessor (such as a VFP coprocessor), but that no corresponding VFP hardware is present in the system. In addition, it might be that the VFP hardware cannot handle the particular instruction and wants to call software to emulate it. Alternatively, the VFP hardware is disabled, and we take the exception so that we can enable it and re-execute the instruction.

Such emulators are called through the undefined instruction vector. They examine the instruction opcode that caused the exception and determine what action to take (for example, perform the appropriate floating-point operation in software). In some cases, such handlers might have to be daisy-chained together, for example, there might be multiple coprocessors to emulate.

If there is no software making use of undefined or coprocessor instructions, the handler for the exception should record suitable debug information and kill the application that failed because of this unexpected event.

An additional use for the undefined instruction exception in some cases is to implement user breakpoints, see Debug for more information on breakpoints. (See also the description of the Linux context switch for VFP in Floating-Point.)

12.4.3 SVC exception handling

A supervisor call (svc) is typically used to permit User mode code to access OS functions. For example, if user code wants to access privileged parts of the system (for example to perform file I/O) it will typically do this using an svc instruction.

Parameters can be passed to the svc handler either in registers or (less frequently) by using the comment field within the opcode.

An example of the use of the svc instruction can be seen by application developers. Tools developed by ARM use `svc 0x123456` (ARM state) or `svc 0xAB` (Thumb) to represent semihosting debug functions (for example, outputting a character on a debugger window).

12.5 External interrupt requests

All microprocessors must respond to external asynchronous events, such as a button being pressed, or a clock reaching a certain value. Normally, there is specialized hardware that activates input lines to the core. This causes the core to temporarily stop the current program sequence and execute a special privileged handler routine. The speed that a core can respond to such events might be a critical issue in system design, and is called interrupt latency. Indeed in many embedded systems, there is no main program. All the functions of the system are handled by code that runs from interrupts, and assigning priorities to these is a key area of design. Rather than the core constantly testing flags from different parts of the system to see if there is something to be done, the system informs the core that something has to happen, by generating an interrupt. Complex systems have many interrupt sources with different levels of priority and requirements for nested interrupt handling in which a higher priority interrupt can interrupt a lower priority one.

Older versions of the ARM architecture allowed implementers considerable freedom in the design of an external interrupt controller, with no agreement over the number or types of interrupts or the software model to be used to interface to the interrupt controller block. The GIC architecture provides a much more tightly controlled specification, with a greater degree of consistency between interrupt controllers from different manufacturers. This enables interrupt handler code to be more portable.

Types of exception, describes how all ARM processors have two external interrupt requests, FIQ and IRQ. Both of these are level-sensitive active-LOW inputs. Individual implementations have interrupt controllers that accept interrupt requests from a wide variety of external sources and map them onto FIQ or IRQ, causing the processor to take an exception.

In general, an interrupt exception can be taken only when the appropriate CPSR disable bit (the F and I bits respectively) is clear.

The `CPS` instruction provides a simple mechanism to enable or disable the exceptions controlled by CPSR A, I and F bits (asynchronous abort, IRQ and FIQ respectively). `CPS` can be used additionally to change mode, as shown below.

```
CPS #<mode>
CPSIE <aif>{, mode}
CPSID <aif>{, mode}
```

where `<mode>` is the number of the mode to change to. If this option is omitted, no mode change occurs. The values of these modes are listed in ARM Processor modes and Registers.

IE or ID will enable or disable exceptions respectively. The exceptions to be enabled or disabled are specified using one or more of the letters A, I and F. Exceptions whose corresponding letters are omitted will not be modified.

In Cortex-R series processors, it is possible to configure the processor so that FIQs cannot be masked by software. This is known as Non-Maskable FIQ and is controlled by a hardware configuration input signal that is sampled when the processor is reset. They will still be masked automatically on taking an FIQ exception.

12.5.1 Assigning interrupts

A system will always have an interrupt controller that accepts interrupt requests from multiple pieces of external hardware. This typically contains a number of registers enabling software running on the core to mask individual interrupt sources, to acknowledge interrupts from external devices and to determine that interrupt sources are currently requesting attention or require servicing.

This interrupt controller can be a design specific to the system, or it can be an implementation of the ARM Generic Interrupt Controller (GIC) architecture, described in The Generic Interrupt Controller.

The Cortex-R4 and Cortex-R5 processors have an interface that enables the core to work with a Vectored Interrupt Controller. In addition to signalling the interrupt to the processor the Vectored Interrupt Controller also provides the address in memory for the interrupt Service Routine. Historically this could speed up the number of cycles in which the core could start the interrupt service routine for the interrupt, because the core could branch directly to the correct handler routine. However, with more recent implementations of the processors it would be less typical to use a Vectored Interrupt Controller as frequency advances in silicon technology mean that it is possible to run the combination of the core and the Generic Interrupt Controller significantly faster than the core and the Vectored Interrupt Controller. This means that typically the Generic Interrupt Controller can provide comparable performance to the Vectored Interrupt Controller but can provide considerably more interrupt lines.

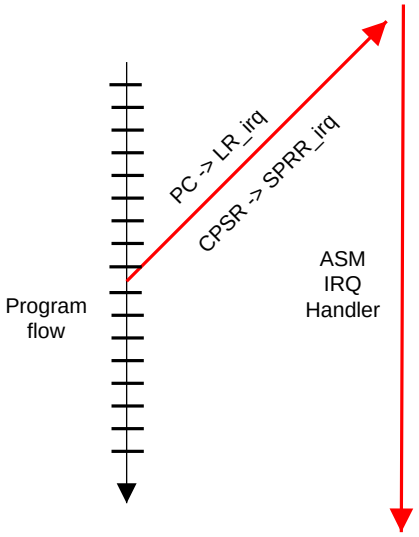
12.5.2 Simplistic interrupt handling

This represents the simplest kind of interrupt handler. On taking an interrupt, additional interrupts of the same kind are disabled until explicitly enabled later. We can only handle additional interrupts at the completion of the first interrupt request and there is no scope for a higher priority or more urgent interrupt to be handled during this time. Such an interrupt handler is described as nonreentrant.This is not generally suitable for complex embedded systems, but it is useful to examine before proceeding to a more realistic example.

The steps taken to handle an interrupt are as follows:

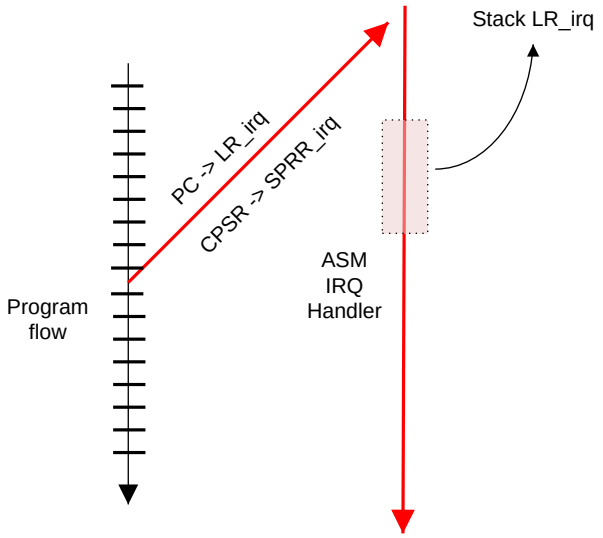
1. An IRQ exception is raised by external hardware. The core performs several steps automatically:
   - The contents of the PC in the current execution mode are stored in LR_IRQ.
   - The CPSR register is copied to SPSR_IRQ.
   - The CPSR content is updated so that the mode bits reflects the IRQ mode, and the I bit is set to mask additional IRQs.
   - The PC is set to the IRQ entry in the vector table.

Figure 12-3: Save the context of the program



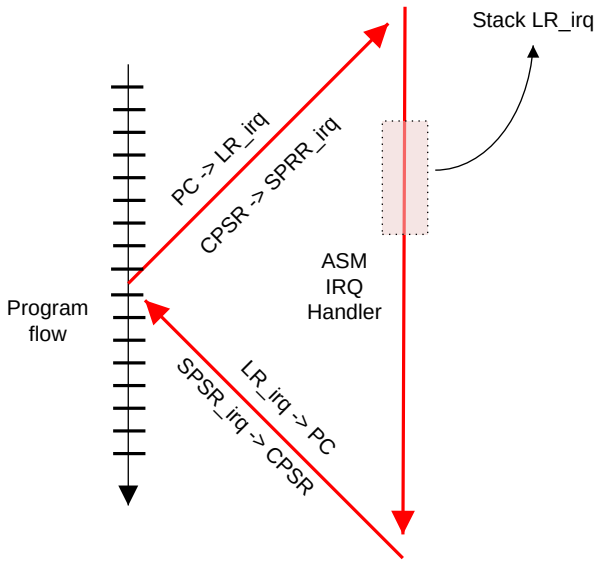
1. The instruction at the IRQ entry in the vector table (a branch to the interrupt handler) is executed.
2. The interrupt handler saves the context of the interrupted program, that is, it pushes onto the stack any registers that will be corrupted by the handler. These registers are popped from the stack when the handler finishes execution.

Figure 12-4: Push registers onto the stack



3. The interrupt handler determines which interrupt source must be processed and calls the appropriate service routine.
4. Prepare the core to switch to previous execution state by copying the SPSR_IRQ to CPSR, and restoring the context saved earlier, and finally the PC is restored from LR_IRQ.

Figure 12-5: Restore the context

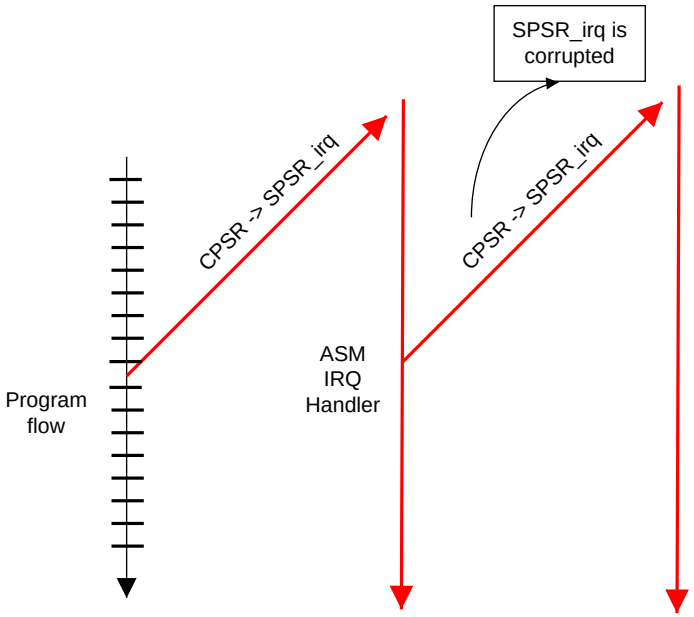


12.5.3 Nested interrupt handling

Nested interrupt handling is where the software is prepared to accept another interrupt, even before it finishes handling the current interrupt. This enables you to prioritize interrupts and make significant improvements to the latency of high priority events at the cost of additional complexity. It is worth noting that nested interrupt handling is a choice made by the software, by virtue of interrupt priority configuration and interrupt control, rather than imposed by hardware.

A reentrant interrupt handler must save the IRQ state and then switch core mode, and save the state for the new core mode, before it branches to a nested subroutine or C function with interrupts enabled. This is because a fresh interrupt could occur at any time, which would cause the core to store the return address of the new interrupt and overwrite the original interrupt return address in the Link Register. When the original interrupt attempts to return to the main program, it will cause the system to fail. The nested handler must switch to an alternative kernel mode before re-enabling interrupts to prevent this.

Figure 12-6: Nested interrupts



Note

A computer program is reentrant if it can be interrupted in the middle of its execution and then be called again before the previous instance has completed.

In Figure 12-6: Nested interrupts on page 143 the value of SPSR must be preserved before interrupts are re-enabled. If it is not, any new interrupt will overwrite the value of SPSR_irq. The

solution to this is to stack the SPSR before re-enabling the interrupts by using the following (which also saves LR):

```
SRSFD  sp!, #0x12
```

Additionally, using the BL instruction within the interrupt handler code will overwrite LR_IRQ. The solution is to switch to Supervisor mode before using the BL instruction.

A reentrant interrupt handler must therefore take the following steps after an IRQ exception is raised and control is transferred to the interrupt handler in the way previously described.

1. The interrupt handler saves the context of the interrupted program (that is, it pushes onto the SVC mode stack any registers that will be corrupted by the handler, including the return address and SPSR_IRQ).
2. It determines which interrupt source must be processed and clears the source in the external hardware (preventing it from immediately triggering another interrupt).
3. The interrupt handler changes to SVC mode, leaving the CPSR I bit set (interrupts are still disabled).
4. The interrupt handler saves the exception return address on the SVC stack and re-enables interrupts.
5. It calls the appropriate handler code.
6. On completion, the interrupt handler disables IRQ and pops the exception return address from the stack.
7. It restores the context of the interrupted program directly from the SVC mode stack. This includes restoring the PC, and the CPSR which switches back to the previous execution mode. If the SPSR does not have the I bit set then the operation also re-enables interrupts.

Sample code for a nested interrupt handler (for non-vectored interrupts) is given below:

```
IRQ_Handler
  SUB        lr, lr, #4
  SRSFD      sp!, #0x1f      @ use SRS to save LR_irq and SPSR_irq in one step onto
 the
                             @ SVC mode stack
  CPS        #0x1f           @ Use CPS to switch to SVC mode

  PUSH       {r0-r3, r12}    @ Store remaining AAPCS registers on the System mode
 stack
  AND        r1, sp, #4      @ Ensure stack is 8-byte aligned. Store adjustment and
                             @ LR_svc to stack
  SUB        sp, sp, r1
  PUSH       {r1, lr}
  BL                         @ identify_and_clear_source
  CPSIE       i              @ Enable IRQ with CPS
  BL          C_irq_handler
  CPSID       i              @ Disable IRQ with CPS
  POP         {r1, lr}       @ Restore LR_svc
  ADD         sp, sp, r1     @ Unadjust stack
  POP         {r0-r3, r12}   @ Restore AAPCS registers
  RFEFD       sp!            @ Return using RFE from the SVC mode stack.
```

12.6 Low latency interrupts

Interrupt latency is the time between the arrival of an interrupt and the start of the corresponding Interrupt Service Routine (ISR).

Cortex-R processors contain a number of features that provide deterministic timing and low interrupt latency for hard real-time applications. These features are collectively referred to as Low Latency Interrupt features.

The low latency interrupt mode can increase the entry speed into an ISR at the cost of a slight reduction in global processor performance.

When the low latency interrupt mode is disabled, all instructions in the pipeline must finish their execution before starting to execute new instructions from the interrupt handler. Some instructions may take considerable time to execute and this can have an effect on interrupt latency if they cannot be interrupted.

When the low latency interrupt mode is enabled, instructions are flushed if they can be restarted without any side effects. These instructions can include:

- All loads and stores that have not started.
- Loads and stores to normal memory that have already started.
- `FDIV` and `FSQRT` operations.
- Certain cache maintenance operations.
- Any pending `DMB` or `DSB` operations
- Instructions that follow these that are already in the pipeline.

Loads and stores to Strongly Ordered or Device memory regions cannot be flushed from the pipeline in the Cortex-R4 and Cortex-R5 processors. In the case of the Cortex-R7 processor with low latency interrupts enabled, accesses to Device or Strongly ordered memory can be flushed from the pipeline until they reach the Load Store Unit stage.

`LDM` and `STM` accesses to Strongly Ordered or Device memory regions could significantly delay the entry into the interrupt service routine as they cannot be abandoned when they have been issued. Where possible, you should try to limit the use of `LDM` and `STM` accesses to Strongly Ordered or Device memory regions.

In the Cortex-R5 processor accesses to the peripheral ports cannot be abandoned when they have started. So if possible, you should limit the use of `STM` and `LDM` instructions to the peripheral port address range.

In the Cortex-R7 processor the peripheral port address region must be defined as Strongly-ordered or device memory. So again, it is best to try and limit `STM` and `LDM` instructions to this address range.

12.7 The Generic Interrupt Controller

The GIC architecture defines a Generic Interrupt Controller (GIC) that comprises a set of hardware resources for managing interrupts in a single or multi-core system. The GIC provides memory-mapped registers that can be used to manage interrupt sources and behavior and (in multi-core systems) for routing interrupts to individual cores. It enables software to mask, enable and disable interrupts from individual sources, to prioritize (in hardware) individual sources and to generate software interrupts. The GIC accepts interrupts asserted at the system level and can signal them to each core it is connected to, potentially resulting in an IRQ or FIQ exception being taken.

From a software perspective, a GIC has two major functional blocks:

Distributor

to which all interrupt sources in the system are wired. The distributor has registers to control the properties of individual interrupts such as priority, state, security, routing information and enable status. The distributor determines which interrupt is to be forwarded to a core, through the attached CPU interface.

CPU Interface

through which a core receives an interrupt. The CPU interface hosts registers to mask, identify and control states of interrupts forwarded to that core. There is a separate CPU interface for each core in the system.

Interrupts are identified in the software by a number, called an interrupt ID. An interrupt ID uniquely corresponds to an interrupt source. Software can use the interrupt ID to identify the source of interrupt and to invoke the corresponding handler to service the interrupt. The exact interrupt ID presented to the software is determined by the system design,

Interrupts can be of a number of different types:

Software Generated Interrupt (SGI)

This is generated explicitly by software by writing to a dedicated distributor register, the Software Generated Interrupt Register (ICDSGIR). It is most commonly used for inter-core communication. SGIs can be targeted at all, or a selected group of cores in the system. Interrupt numbers 0-15 are reserved for this. The exact interrupt number used for communication is at the discretion of software.

Private Peripheral Interrupt (PPI)

This is generated by a peripheral that is private to an individual core. Interrupt numbers 16-31 are reserved for this. These identify interrupt sources private to the core, and is independent of the same source on another core, for example, per-core timer.

Shared Peripheral Interrupt (SPI)

This is generated by a peripheral that the Interrupt Controller can route to more than one core. Interrupt numbers 32-1020 are used for this. SPIs are used to signal interrupts from various peripherals accessible across the whole the system.

Interrupts can either be edge-triggered (considered to be asserted when the Interrupt Controller detects a rising edge on the relevant input - and to remain asserted until cleared) or level-sensitive (considered to be asserted only when the relevant input to the Interrupt Controller is HIGH).

An interrupt can be in a number of different states:

- Inactive - this means that the interrupt is not asserted yet.
- Pending - this means that the interrupt source has been asserted, but is waiting to be handled by a core. Pending interrupts are candidates to be forwarded to the CPU interface and then later on to the core.
- Active - this describes an interrupt that has been acknowledged by a core and is currently being serviced.
- Active and pending - this describes the situation where a core is servicing the interrupt and the GIC also has a pending interrupt from the same source.

The priority and list of cores to which an interrupt can be delivered to are all configured in the distributor. An interrupt asserted to the distributor by a peripheral will be in the Pending state (or Active and Pending if was already Active). The distributor determines the highest priority pending interrupt that can be delivered to a core and forwards that to the CPU interface of the core. At the CPU interface, the interrupt is in turn signalled to the core, at which point the core takes the FIQ or IRQ exception.

The core executes the exception handler in response. The handler must query the interrupt ID from a CPU interface register and begin servicing the interrupt source. When finished, the handler must write to a CPU interface register to report the end of processing. Later on the CPU interface is prepared to signal the next interrupt forwarded to it by the distributor.While servicing an interrupt, the distributor cycles through Pending, Active states, ending in Inactive state when it has finished. The state of an interrupt is therefore reflected in the distributor registers.

12.7.1 Configuration

The GIC is accessed as a memory-mapped peripheral. All cores can access the common distributor block, but the CPU interface is banked, that is, each core uses the same address to access its own private CPU interface. It is not possible for a core to access the CPU interface of another core.

The distributor hosts a number of registers that you can use to configure the properties of individual interrupts. These configurable properties are:

- An interrupt priority. The distributor uses this to determine which interrupt is next forwarded to the CPU interface.
- An interrupt configuration. This determines if an interrupt is level- or edge-sensitive.
- An interrupt target. This determines a list of cores to which an interrupt can be forwarded.
- Interrupt enable or disable status. Only those interrupts that are enabled in the distributor are eligible to be forwarded when they become pending.
- Interrupt security determines whether the interrupt is allocated to Secure or Normal world software.
- An Interrupt state.

The distributor also provides priority masking by which interrupts below a certain priority are prevented from reaching the core. The distributor uses this when determining whether a pending interrupt can be forwarded to a particular core.

The CPU interfaces on each core helps with fine-tuning interrupt control and handling on that core.

12.7.2 Initialization

Both the distributor and the CPU interfaces are disabled at reset.The GIC must be initialized after reset before it can deliver interrupts to the core.

In the distributor, software must configure the priority, target, security and enable individual interrupts. The distributor block must subsequently be enabled through its control register. For each CPU interface, software must program the priority mask and preemption settings.

Each CPU interface block itself must be enabled through its control register. This prepares the GIC to deliver interrupts to the core.

Before interrupts are expected in the core, software prepares the core to take interrupts by setting a valid interrupt vector in the vector table, and clearing interrupt masks bits in the CPSR.

The entire interrupt mechanism in the system can be disabled by disabling the distributor block. Interrupt delivery to an individual core can be disabled by disabling its CPU interface block, or by setting mask bits in CPSR of that core. Individual interrupts can also be disabled (or enabled) in the distributor.

For an interrupt to reach the core, the individual interrupt, distributor and CPU interface must all be enabled, and the CPSR interrupt mask bits cleared.

12.7.3 Interrupt handling

When the core takes an interrupt, it jumps to the top-level interrupt vector obtained from the vector table and begins execution.

The top-level interrupt handler reads the Interrupt Acknowledge Register from the CPU Interface block to obtain the interrupt ID.

In addition to returning the interrupt ID, the read causes the interrupt to be marked as active in the distributor. When the interrupt ID is known (identifying the interrupt source), the top-level handler can now dispatch a device-specific handler to service the interrupt.

When the device-specific handler finishes execution, the top-level handler writes the same interrupt ID to the End of Interrupt register in the CPU Interface block, indicating the end of interrupt processing.

Apart from removing the active status, which will make the final interrupt status either Inactive, or Pending (if the state was Active and Pending), this will enable the CPU Interface to forward more pending interrupts to the core. This concludes the processing of a single interrupt.

It is possible for there to be more than one interrupt waiting to be serviced on the same core, but the CPU Interface can signal only one interrupt at a time. The top-level interrupt handler repeats the sequence until it reads the special interrupt ID value 1023, indicating that there are no more interrupts pending at this core. This special interrupt ID is called the spurious interrupt ID.

The spurious interrupt ID is a reserved value, and cannot be assigned to any device in the system. When the top-level handler has read the spurious interrupt ID it can complete its execution, and prepare the core to resume the task it was doing before taking the interrupt.

13. Fault Detection and Control Features

In silicon devices, stray radiation and other effects can cause the data stored in RAM to be corrupted. Error detection and correction techniques can be used to help mitigate the effect of such errors. The Cortex-R processors include features that provide a means of detecting some of these errors, potentially correcting the incorrect value and alerting the processor or the system to the event so that corrective or protective action can be taken in a predictable manner, making them suitable for use in safety-related systems.

13.1 Types of errors

The fault detection and control features of the Cortex-R processors are primarily aimed at bit errors in TCMs or level one caches. A bit error refers to an incorrect binary digit in a RAM chunk. For example, an alpha particle strike can invert one or more bits of data stored in RAM.

Errors can be classified into:

- Soft errors.
- Hard errors.
- Fatal errors.

Soft errors are errors that do not persist. A new value can be written to the RAM and read back correctly. They are usually caused by interference but could also be because of hardware failure. The Cortex-R processors can detect and correct soft errors.

Soft errors are an increasing concern in modern systems. Smaller transistor geometries and lower voltages give circuits an increased sensitivity to perturbation by cosmic radiation, alpha particles from silicon packages, electrical noise, or other background radiation. This is particularly true for memory devices that rely on storing small amounts of charge and that also occupy large proportions of total silicon area. Without sufficient protection against soft errors, the mean time between failure could be seconds.

Hard errors are bit errors that persist even after correction. These are usually because of a hardware failure of the RAM circuit. For example, a faulty RAM location might always read high for a particular data bit. The processor is not able to correct this error as it cannot consistently write the correct data to the RAM location. However, the safety features enable the processor to recognise the presence of hard errors and mitigate the effects.

A fatal error is one in which the processor is not able to recover the correct data value, either through error correction or through a memory access. The implication here is that the corrupted data value has the potential to cause a program to execute incorrectly.

For safety-related applications, a fatal error should be considered serious. For safety critical applications, the solution to recover from a fatal error might be:

- Reloading a memory.

- Restarting a subroutine.
- Resetting the processor.

However, in applications that are not safety critical, fatal errors can be ignored. An example is when driving an output, that is fault tolerant, such as a display.

Bit errors can also be classified into:

- Correctable errors.
- Non-correctable errors.

A bit error is correctable if the original data can be recovered using only the ECC bits. A non-correctable error is a bit error where the original data cannot be recovered using the ECC bits alone. A non-correctable error is not a fatal error if the correct data can be fetched from another memory.

13.2 Error detection methods

The Cortex-R processors implement two methods of error checking in caches and TCMs:

- Parity.
- Error Checking and Correction.

The Cortex-R processors implement one method of error detection in processor operation.

- Redundant logic.

13.2.1 Parity

The parity method is a simple method to check single bit errors. It uses an additional bit to mark whether there is an even or odd number of bits with value 1. The additional bit is called the parity bit. A parity bit can be allocated per data chunk, typically this data chunk might be a byte or a word.

With even parity the value of the parity bit is set so that there is an even number of [1] bits in the chunk of data. With odd parity the value of the parity bit is set so that there is an odd number of [1] bits in the chunk of data.

When the data and parity bit are read back from the memory the total number of [1] bits are checked. If the number of [1] bits does not match the parity settings then there is an error in the data. It is not possible to identify which bit of data is faulty.

Parity checking is only able to detect an odd number of faulty bits, if an even number of bits have incorrect values the parity check will not detect the error.

With parity checking enabled, the Cortex-R processors normally store one parity bit per byte. This makes it possible to detect errors in individual bytes.

Parity is checked on reads and writes, and can be implemented on both tag and data RAMs (and TCMs, in the case of the Cortex-R4 processor). Parity mismatch generates a prefetch or data abort exception, and the fault status address registers are updated appropriately.

When a parity error is detected it is not possible to determine which bit is incorrect and so the value in the RAM cannot be corrected. However, if the error is detected in a clean cache line then it is still possible to recover from the error by invalidating the line and re-fetching the relevant data from the external memory.

13.2.2 Error Checking and Correction

Error Checking and Correction (ECC) is a method of detecting and correcting one or more bit errors in a chunk of data. All the Cortex-R class ECC schemes can correct a single bit error and can detect when there are two bit errors but will not be able to correct the two bit errors. As a result these are called Single-bit Error-Correction, Double -bit Error Detection (SEC-DED) ECC schemes.

When the SEC-DED ECC scheme is used to protect the RAMs several additional bits of data are saved for each chunk of data. This could be for only a few bits of data or perhaps for a 64-bit double word. For each data chunk a number of redundant code bits are computed and stored with the data. When the processor reads the data it can detect up to two errors in the data chunk or its code bits. It can correct any single error in the data chunk or its associated code bits.

If there are more than two errors in a data chunk and its associated code bits, they might or might not be detected. The error scheme might interpret such a condition as a single error and make an unsuccessful attempt at a correction.

Recovery calculations can take several cycles.

ECC schemes require more redundant data than a simple parity check, for the SEC-DED ECC scheme used in the Cortex-R processors:

- 3 bits data requires 4 bits ECC code.
- 32 bits data requires 7 bits ECC code.
- 64 bits data requires 8 bits ECC code.

If the full data width covered by ECC is not provided in a store operation, the processor must do an additional read before it can calculate the new ECC value. This is called a read-modify-write operation.

When a single bit ECC error is detected, Cortex-R series processors can correct the value in the RAM. When a double bit ECC error is detected in a clean cache line, the processors can often recover from the error by invalidating the line and re-fetching the relevant data from the external memory. However if a double-bit ECC error is detected in a dirty cache-line or TCM, the processor will not be able to fully recover from the error. The core must analyze the source of the issue and determine whether any protective action must be taken. This could involve disabling the caches or even initiating a shutdown of the processor.

13.2.3 ECC and parity initialization

Both ECC and Parity require the full data chunk in order to correctly calculate the parity or ECC bits. When the location is updated with an access smaller than the data chunk the processor needs to know the values of the current data chunk in order to correctly update the parity or ECC bits. Typically this achieved with a Read-Modify-Write approach: the data chunk is first read, this data is updated with the new data, the new parity or ECC bits are calculated and then the updated data is written back.

When a RAM first comes out of reset the RAM values will have random values and so this read-modify-write approach will not work. This means that the first write to a RAM must be for the full data chunk to ensure that the parity or ECC bits are generated correctly.

13.2.4 Redundant logic

The redundant logic safety feature replicates an entire block of logic or replicates the entire core in the design. Both the primary logic and the redundant logic are driven with exactly the same code and data values. The outputs from the blocks are compared. If there is a difference in the outputs it indicates that an error has occurred somewhere.

The redundant core performs the same work as the primary core but operates a number of clock cycles behind the master core. This mode of operation is called lock-step.

When there is a mismatch between the outputs of the primary logic and the redundant logic, the processor or an external monitor must determine what protective action is required.

13.3 Error signalling

When an error is detected it might be necessary to signal to the software that the error has occurred. The processor would then analyze the error and determine what corrective or protective action is required.

13.3.1 No signal

When an error is fully corrected by the hardware it might not be necessary to signal to the processor that the error has occurred. It is likely that the design monitors such errors. If there were a significant number of such errors, it might indicate a systemic issue.

13.3.2 Abort

The processor could be designed to take an abort exception when an error is detected. When the abort exception occurs the abort handler identifies that the abort was because of the detection of an error and will then determine what corrective or protective action is required.

Using an abort exception ensures that the processor is able to analyze the error quickly and will help ensure that the processor does not execute instructions with incorrect data. However this could be disruptive to the normal flow of operation, especially if an abort is signalled for a correctable error.

13.3.3 Interrupt

The processor could generate an output event or interrupt when an error is detected. This can then be used to trigger an interrupt to the processor. When the processor takes the interrupt, the interrupt handler determines that the interrupt was because of the detection of an error and then determines what corrective or protective action is required.

Using an interrupt to signal the error is likely to be less disruptive to the normal flow of operation of the processor as the interrupts can typically be prioritized in an interrupt controller so that the signalling of the error can be prioritized accordingly. However, this does mean that the processor might execute instructions with incorrect data if the interrupt for a non-correctable error is not able to interrupt the program flow quickly enough.

13.4 Recovering from hard errors

If there is a hard error in a RAM, for example if a bit becomes permanently stuck at 1 or 0, then this RAM location can no longer be used to correctly store data, because ECC and Parity protection schemes cannot recover from such errors.

The presence of hard errors can adversely affect the behavior of ECC or parity logic. If an error is detected during a cache look-up or when data is read from a TCM then this data must be corrected before it is used by the processor. When the processor attempts to correct the data it is unable to do so as it cannot change the value of the stuck bit. When the processor attempts the access again, it detects the same error and again attempts to correct the issue. It is possible for a processor to enter a live-lock loop as it continually attempts to correct and then re-read the data.

To guard against live-lock, the processor might include error banks to mask the location that contains the error. After the processor has first detected and attempted to correct the error, the RAM location is saved in a register bank. When the processor retries the access it finds the location in the error bank and either reads the correct data from the register bank itself or from external memory.

Alternatively the processor might guard against live-lock by monitoring repeated accesses to the same location and signalling an event if the same location is accessed continuously.

13.5 Power and performance

The use of parity, ECC protection, or redundant logic will have a cost in terms of power consumption and area because of the additional logic that is required, and because the use of parity and ECC logic restricts writes to the cache or TCM. There is a relatively low percentage increase in power and area for parity and ECC protection. When using redundant logic it is possible to replicate the entire processor or only a few parts of the processor in the design.

Whenever a write is made to the cache or TCM, the ECC or parity bits must be calculated and updated in the RAM. To update the parity or ECC bits correctly the full chunk of data must be written to the RAM. If the parity or ECC bits are calculated on a full word or doubleword, then to update a byte in that chunk of data, using the read-modify-write process, the processor must:

1. Read the current data from the cache.
2. Calculate the new parity or ECC.
3. Write the full data back to RAM.

These additional reads from the caches or TCMs can affect the performance of code that contains a significant number of byte or half word memory accesses.

The use of redundant logic can have an effect on performance. This is because the maximum frequency that can be achieved by the primary processor can be restricted by:

- Sharing input signals to each block of logic.
- Capture and analysis of the output signals from each block of logic.

13.6 Fault detection and control features in the Cortex-R4 processor

Each of the Cortex-R series processors contain a number of features that enhance their suitability for safety-related applications. Semiconductor manufacturers may choose, or not choose, to implement these fault detection features in their processor implementation. These should not be assumed to be present in all Cortex-R devices

The Cortex-R4 processor provides the following optional fault detection features:

- Parity in Cache RAM in the Cortex-R4 processor
- ECC for the Cache RAM in the Cortex-R4 processor
- Parity for the TCM in the Cortex-R4 processor
- ECC for the TCMs in the Cortex-R4 processor
- Hard error banks in the Cortex-R4 processor
- Bus protection on the Cortex-R4 processor

- Redundant core in the Cortex-R4 processor
- Test of the fault detection and control features on the Cortex-R4 processor

Parity in Cache RAM in the Cortex-R4 processor

Parity bit generation and checking can be configured for the cache RAMs in the Cortex-R4 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bits[8:7] = 0b01 Parity on icache
                           ; bits[6:5] = 0b01 Parity on dcache
```

Enabling parity protection forces the caches to operate in write-through mode. Regions of memory marked as write-back are treated as write-through memory. This ensures that the core can recover from soft-parity errors. On detection of a parity error the processor invalidates the relevant cache line and then re-fetches the line from external memory.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b101
BIC r1, r1, #(0x1 << 4)      ; to enable parity
ORR r1, r1, #(0x1 << 3)
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

The processor can be configured to generate an abort when a parity error occurs. The abort handler can then read the correctable fault location register and determine whether to take any protective action.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
BIC r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b000
BIC r1, r1, #(0x1 << 4)      ; to enable parity and force an
BIC r1, r1, #(0x1 << 3)      ; abort on all parity detected errors
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR

.........

MRC p15, 0, r0, c15, c3, 0  ; Read CFLR
```

Parity errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitoring Unit for analysis.

```
MOV r0, #0                  ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4D               ; Data cache data RAM parity error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.6.1 ECC for the Cache RAM in the Cortex-R4 processor

ECC generation and checking can be configured for the cache RAMs in the Cortex-R4 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bits[8:7] = 0b11 ECC on icache
                           ; bits[6:5] = 0b10 ECC on dcache
```

When enabling ECC it is possible to force the caches to operate in write-through mode. Write-back memory regions are treated as write-through memory. This ensures that the core can recover from soft ECC errors. On detection of an ECC error the processor invalidates the relevant cache line and then re-fetches the line from external memory.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b110
BIC r1, r1, #(0x1 << 4)      ; to enable ECC with forced
ORR r1, r1, #(0x1 << 3)      ; write-through
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

It is also possible to use ECC with write-back memory regions. When configured to use write-back memory, the core can recover from some ECC errors.

If hardware recovery is enabled then:

- On detection of an ECC error in a clean cache line the processor invalidates the line and then re-fetches the line from external memory.
- On detection of a single-bit error in a dirty cache line the line is evicted and corrected in-line before being written back to main memory, the line is invalidated and the corrected data re-fetched from external memory.
- On detection of a double-bit error in a dirty cache line the line is evicted and written back to main memory. As the error cannot be corrected, the word or words containing the double bit error will not be written back to main memory. The line is invalidated and an abort is generated by the processor so that the error can be analyzed.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b101
BIC r1, r1, #(0x1 << 4)      ; to enable ECC no forced
ORR r1, r1, #(0x1 << 3)      ; write-through
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

The processor can be configured to generate an abort whenever an ECC error occurs. The abort handler would then read the correctable fault location register and determine whether any protective action should be taken. (Double-bit errors will always generate aborts.)

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b010
BIC r1, r1, #(0x1 << 4)      ; to enable ECC with forced
ORR r1, r1, #(0x1 << 3)      ; write-through. Generates abort
                            ; on error detection.
```

```
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
BIC r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b000
BIC r1, r1, #(0x1 << 4)      ; to enable ECC no forced
BIC r1, r1, #(0x1 << 3)      ; write through. Generates abort
MCR p15, 0, r1, c1, c0, 1  ; Write ACTLR

.........

MRC p15, 0, r0, c15, c3, 0  ; Read CFLR
```

ECC errors are signalled with events and so these events can be captured and monitored by external hardware or the Performance Monitor unit and used for analysis.

```
MOV r0, #0                 ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4D               ; Data cache data RAM correctable ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB

.............................

MOV r0, #1                 ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x60               ; Data cache data RAM fatal ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.6.2 Parity for the TCM in the Cortex-R4 processor

Parity bit generation and checking can be configured for the TCMs in the Cortex-R4 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1   ; read Build Options 2 Register
                             ; bits[27:26] = 0b01 Parity on ATCM
                             ; bits[25:24] = 0b01 Parity on BTCM

MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 27)     ; Enable parity detection on B1TCM port
ORR r1, r1, #(0x1 << 26)     ; Enable parity detection on B0TCM port
ORR r1, r1, #(0x1 << 25)     ; Enable parity detection on ATCM port
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR
```

On detection of a parity error, the processor generates an abort. The abort handler must determine an appropriate corrective or protective action.

```
MRC p15, 0, r0, c5, c0, 0  ; Read DFSR, DFSR[10,3:1] = 0b1100 indicates a
                           ; parity error

.........................

MRC p15, 0, r0, c5, c0, 1  ; Read IFSR, IFSR[10,3:1] = 0b1100 indicates a
                           ; parity error
```

Parity errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                 ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4E              ; Data cache data RAM fatal ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.6.3 ECC for the TCMs in the Cortex-R4 processor

ECC generation and checking can be configured for the TCM RAM in the Cortex-R4 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1   ; read Build Options 2 Register
                             ; bits[27:26] = 0b10 32 bit ECC on ATCM
                             ; bits[25:24] = 0b10 32 bit ECC on BTCM
                             ; bits[27:26] = 0b11 64 bit ECC on ATCM
                             ; bits[25:24] = 0b10 64 bit ECC on BTCM
```

On detection of a single-bit error the processor corrects the data and writes it back to the TCM. The processor then re-reads the corrected data from the address location.

```
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 27)     ; Enable parity detection on B1TCM port
ORR r1, r1, #(0x1 << 26)     ; Enable parity detection on B0TCM port
ORR r1, r1, #(0x1 << 25)     ; Enable parity detection on ATCM port
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR
```

Alternatively, instead of correcting the data, the processor could be configured to take an abort. The abort handler must determine the appropriate corrective or protective action.

```
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 27)     ; Enable parity detection on B1TCM port
ORR r1, r1, #(0x1 << 26)     ; Enable parity detection on B0TCM port
ORR r1, r1, #(0x1 << 25)     ; Enable parity detection on ATCM port
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR

MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
ORR r1, r1, #(0x1 << 3)      ; Disable parity detection on BTCM
ORR r1, r1, #(0x1 << 2)      ; Disable parity detection on BTCM
MCR p15, 0, r1, c1, c0, 1    ; Write Secondary Auxiliary Control Register
```

On detection of a double-bit error the data cannot be corrected automatically and so the processor takes an abort. The abort handler must determine the appropriate corrective or protective action.

```
MRC p15, 0, r0, c5, c0, 0    ; Read DFSR, DFSR[10,3:1] = 0b1100 indicates an
                             ; ECC error

.................
```

```
MRC p15, 0, r0, c5, c0, 1    ; Read IFSR, IFSR[10,3:1] = 0b1100 indicates an
                             ; ECC error
```

ECC errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                   ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5   ; Write PMNXSEL Register
ISB
MOV r1, #0x6B                ; TCM correctable ECC error reported by the PFU
MCR p15, 0, r1, c9, c13, 1   ; Write EVTSELx Register
ISB

.................

MOV r0, #1                   ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5   ; Write PMNXSEL Register
ISB
MOV r1, #0x4E                ; TCM fatal ECC error reported by the PFU
MCR p15, 0, r1, c9, c13, 1   ; Write EVTSELx Register
ISB
```

ECC checking can also be configured externally to the Cortex-R4, as part of the TCM RAM instance. If an error is detected by this external TCM logic an error is signalled to the Cortex-R4 processor, if the error is uncorrectable the processor will take an abort.

```
MRC p15, 0, r0, c5, c0, 0    ; Read DFSR, DFSR[10,3:1] = 0b1100 indicates an
                             ; External synchronous abort

................

MRC p15, 0, r0, c5, c0, 1    ; Read IFSR, IFSR[10,3:1] = 0b1100 indicates an
                             ; External Synchronous Abort
```

13.6.4 Hard error banks in the Cortex-R4 processor

The Cortex-R4 processor can be configured to include a hard error bank for the TCMs. The error bank helps protect the TCM error correction logic against livelocks, which can occur in the presence of hard errors.

```
MRC p15, 0, r0, c15, c2, 1   ; read Build Options 2 Register
                             ; bit[4] = 0b0 Includes hard error cache
```

The Cortex-R4 processor includes an error bank for the caches. The error bank helps protect the cache error correction logic against livelocks, which can occur in the presence of hard errors. This bank can be enabled or disabled using the system control registers.

```
MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
BIC r1, r1, #(0x1 << 22)     ; Enable hard-error support in the caches
MCR p15, 0, r1, c1, c0, 0    ; Write Secondary Auxiliary Control Register

.................

MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
```

```
ORR r1, r1, #(0x1 << 22)      ; Disable hard-error support in the caches
MCR p15, 0, r1, c1, c0, 0     ; Write Secondary Auxiliary Control Register
```

The Cortex-R4 processor includes an event to indicate if the processor is in livelock due to hard errors. This event can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #                     ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5    ; Write PMNXSEL Register
ISB
MOV r1, #0x62                 ; Processor livelock
MCR p15, 0, r1, c9, c13, 1    ; Write EVTSELx Register
ISB
```

13.6.5 Bus protection on the Cortex-R4 processor

The Main AXI master interface, AXI slave port and TCM interfaces can be configured with parity generation and checking on the buses.

```
MRC p15, 0, r0, c15, c2, 1    ; read Build Options 2 Register
                              ; bit[10] = 0b1 Includes TCM Bus Parity
                              ; bit[3] = 0b1 Includes AXI Bus Parity
```

The core will generate the parity bits for output signals and will check the parity bits for input signals. If the core detects an error on one of the buses, it will generate an abort.

```
MRC p15, 0, r0, c5, c0, 0     ; Read DFSR, DFSR[10,3:0] = 0b10110 indicates an
                              ; External asynchronous abort

.................

MRC p15, 0, r0, c5, c0, 1     ; Read IFSR, IFSR[10,3:0] = 0b01000 indicates an
                              ; External Synchronous Abort
```

To use the bus parity protection the system design around the core must also include the complementary parity bit generation and checking.

13.6.6 Redundant core in the Cortex-R4 processor

There is an implementation option in the Cortex-R4 processor to include a fully redundant copy of the processor implementation on the silicon. The outputs from the functional and redundant processors are compared with user defined logic to identify any differences in behavior. If a difference is detected then this would be signalled with an event that might be sent to the processor or another logic block for analysis.

In the Cortex-R4 processor, the redundant core typically operates 1.5 cycles behind the primary core.

The exact behavior of the redundant core comparison logic is implementation defined.

```
MRC p15, 0, r0, c15, c2, 1    ; read Build Options 2 Register
                              ; bit[31] = 0b1 Includes Redundant Core.
```

13.6.7 Test of the fault detection and control features on the Cortex-R4 processor

There are two mechanisms to test that the ECC and parity features are working correctly in the Cortex-R4 processor.

The AXI slave provides an interface to access the caches. Using this, it is possible to deliberately induce errors in the caches that can then be detected and corrected when accessed via instructions from the processor.

The AXI slave can be used to preload the TCMs but cannot be used to test errors in the TCMs.

The processor itself can be used to induce errors in the TCMs. ECC checking is enabled and disabled through the CP15. While ECC checking is disabled a sub-word access to a TCM location will not induce a read-modify-write process. This means that by doing sub-word accesses to TCM locations while ECC checking is disabled it is possible to generate a mismatch between the data value and the ECC code.

```
; Setup - these addresses MUST be doubleword aligned
LDR r10,=0x40000000           ; Address of victim TCM location to insert error
LDR r11,=0x40000008           ; Address of scratch memory location

LDR r0,=0xDEAFBEEF
LDR r1,=0xCAFEBABE

STR r0,[r10]
STR r1,[r10,#0x4]

; Disable ECC - ATCM
MRC p15,0,r0,c1,c0,1          ; Read aux ctl
BIC r0,r0,#0x02000000
MCR p15,0,r0,c1,c0,1

; Memory barrier to ensure all previous accesses have completed
DMB

; Read data and introduce ECC error
LDRD r0,[r10]                 ; Read data to corrupt
STRD r0,[r11]                 ; Store data to corrupt in scratch location

LSR  r2, r0, #16              ; Only interested in 3rd byte
EOR  r2, #0x2                  ; Toggle one bit
STRB r2,[r11,#2]              ; Store byte to corrupt data and ECC
LDRD r0,[r11]                 ; Load corrupted data back to set up
                              ; internal registers
BFI  r0, r1, #0, #8           ; Merge byte 0 from upper word into lower word
ROR  r0, r0, #24              ; Shift byte 3 into byte 0 position,
                              ; byte 0 into byte 1 position.
STRH r0, [r10,#3]             ; Store byte - data unchanged, but ECC changed

; Enable ECC
```

```
 MRC p15,0,r0,c1,c0,1          ; Read aux ctl
 ORR r0,r0,#0x02000000
 MCR p15,0,r0,c1,c0,1
                               ; Read data - should get corrected
from 0xDEAFBEEF
                               ; to 0xDEADBEEF
 LDRD r0,[r10]
```

13.7 Fault detection and control features in the Cortex-R5 processor

Each of the Cortex-R series processors contain a number of features that enhance their suitability for safety-related applications. Semiconductor manufacturers may choose, or not choose, to implement these fault detection features in their processor implementation. These should not be assumed to be present in all Cortex-R devices

The Cortex-R5 processor provides the following optional fault detection features:

- Parity in Cache RAM in the Cortex-R5 processor
- ECC for the Cache RAM in the Cortex-R5 processor
- ECC for the TCMs in the Cortex-R5 processor
- Hard error banks in the Cortex-R5 processor
- Bus protection on the Cortex-R5 processor
- Redundant core in the Cortex-R5 processor
- Test of the fault detection and control features on the Cortex-R5 processor

13.7.1 Parity in Cache RAM in the Cortex-R5 processor

Parity bit generation and checking can be configured for the cache RAM in the Cortex-R5 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bits[8:7] = 0b01 Parity on icache
                           ; bits[6:5] = 0b01 parity on dcache
```

Enabling parity protection forces the caches to operate in write-through mode. Regions of memory marked as write-back are treated as write-through memory. This ensures that the core can recover from soft-parity errors. On detection of a parity error the processor invalidates the relevant cache line and then re-fetches the line from external memory.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b101
BIC r1, r1, #(0x1 << 4)      ; to enable parity
ORR r1, r1, #(0x1 << 3)
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

The processor can be configured to generate an abort when a parity error occurs. The abort handler can then read the correctable fault location register and determine whether to take any protective action.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
BIC r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b000
BIC r1, r1, #(0x1 << 4)      ; to enable parity and force an
BIC r1, r1, #(0x1 << 3)      ; abort on all parity detected errors
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR

.........

MRC p15, 0, r0, c15, c3, 0  ; Read CFLR
```

Parity errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                  ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4D               ; Data cache data RAM parity error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.7.2 ECC for the Cache RAM in the Cortex-R5 processor

ECC generation and checking can be configured for the cache RAM in the Cortex-R5 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bits[8:7] = 0b01 ECC on icache
                           ; bits[6:5] = 0b01 ECC on dcache
```

When enabling ECC it is possible to force the caches to operate in write-through mode. Write-back memory regions are treated as write-through memory. This ensures that the core can recover from soft ECC errors. On detection of an ECC error the core invalidates the relevant cache line and then re-fetches the line from external memory.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)      ; Set Bits [5:3] = 0b110
BIC r1, r1, #(0x1 << 4)      ; to enable ECC with forced
ORR r1, r1, #(0x1 << 3)      ; write-through
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

It is also possible to use ECC with write-back memory regions.When configured to use write-back memory, the core can recover from some ECC errors.

If hardware recovery is enabled then:

- On detection of an ECC error in a clean cache line the processor invalidates the line and then re-fetches the line from external memory.

- On detection of a single-bit error in a dirty cache line the line is evicted and corrected in-line before being written back to main memory, the line is invalidated and the corrected data re-fetched from external memory.
- On detection of a double-bit error in a dirty cache line the line is evicted and written back to main memory. As the error cannot be corrected, the word or words containing the double bit error will not be written back to main memory. The line is invalidated and an abort is generated by the processor so that the error can be analyzed.

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)     ; Set Bits [5:3] = 0b101
BIC r1, r1, #(0x1 << 4)     ; to enable ECC no forced
ORR r1, r1, #(0x1 << 3)     ; write-through
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
```

The processor can be configured to generate an abort whenever an ECC error occurs. The abort handler would then read the correctable fault location register and determine whether any protective action should be taken. (Double-bit errors will always generate aborts.)

```
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
ORR r1, r1, #(0x1 << 5)     ; Set Bits [5:3] = 0b010
BIC r1, r1, #(0x1 << 4)     ; to enable ECC with forced
ORR r1, r1, #(0x1 << 3)     ; write-through. Generates abort
                            ; on error detection.
MCR p15, 0, r1, c1, c0, 1   ; Write ACTLR
MRC p15, 0, r1, c1, c0, 1   ; Read ACTLR
BIC r1, r1, #(0x1 << 5)     ; Set Bits [5:3] = 0b000
BIC r1, r1, #(0x1 << 4)     ; to enable ECC no forced
BIC r1, r1, #(0x1 << 3)     ; write through. Generates abort
MCR p15, 0, r1, c1, c0, 1  ; Write ACTLR

.........

MRC p15, 0, r0, c15, c3, 0  ; Read CFLR
```

ECC errors are signalled with events and so these events can be captured and monitored by external hardware or the Performance Monitor unit and used for analysis purposes.

```
MOV r0, #0                 ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4D              ; Data cache data RAM correctable ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB

.............................

MOV r0, #1                 ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x60              ; Data cache data RAM fatal ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.7.3 ECC for the TCMs in the Cortex-R5 processor

ECC generation and checking can be configured for the TCM RAM in the Cortex-R5 processor during implementation.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bits[27:26] = 0b10 32 bit ECC on ATCM
                           ; bits[25:24] = 0b10 32 bit ECC on BTCM
                           ; bits[27:26] = 0b11 64 bit ECC on ATCM
                           ; bits[25:24] = 0b10 64 bit ECC on BTCM
```

When a single-bit error is detected the core corrects the data and writes it back to the TCM. The core then re-reads the corrected data from the address location.

```
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 27)     ; Enable parity detection on B1TCM port
ORR r1, r1, #(0x1 << 26)     ; Enable parity detection on B0TCM port
ORR r1, r1, #(0x1 << 25)     ; Enable parity detection on ATCM port
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR
```

Alternatively, instead of correcting the data, the core can be configured to take an abort. The abort handler must then determine the appropriate corrective or protective action.

```
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 27)     ; Enable parity detection on B1TCM port
ORR r1, r1, #(0x1 << 26)     ; Enable parity detection on B0TCM port
ORR r1, r1, #(0x1 << 25)     ; Enable parity detection on ATCM port
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR

MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
ORR r1, r1, #(0x1 << 3)      ; Disable parity detection on BTCM
ORR r1, r1, #(0x1 << 2)      ; Disable parity detection on BTCM
MCR p15, 0, r1, c1, c0, 1    ; Write Secondary Auxiliary Control Register
```

When a double-bit error is detected the data cannot be corrected automatically and so the core takes an abort. The abort handler must again determine the appropriate corrective or protective action.

```
MRC p15, 0, r0, c5, c0, 0  ; Read DFSR, DFSR[10,3:1] = 0b1100 indicates an
                           ; ECC error
.................
MRC p15, 0, r0, c5, c0, 1  ; Read IFSR, IFSR[10,3:1] = 0b1100 indicates an
                           ; ECC error
```

ECC errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                 ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x6B               ; TCM correctable ECC error reported by PFU
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

```
.............................

MOV r0, #1                  ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x4E               ; Data cache data RAM fatal ECC error
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

ECC checking can also be configured externally to the Cortex-R5, as part of the TCM RAM instance. If an error is detected by this external TCM logic an error is signalled to the Cortex-R5 processor, if the error is uncorrectable the processor will take an abort.

```
MRC p15, 0, r0, c5, c0, 0  ; Read DFSR, DFSR[10,3:0] = 0b10110 indicates an
                           ; External synchronous abort
.............
MRC p15, 0, r0, c5, c0, 1  ; Read IFSR, IFSR[10,3:0] = 0b01000 indicates an
                           ; External synchronous abort
```

13.7.4 Hard error banks in the Cortex-R5 processor

The Cortex-R5 processor can be configured to include a hard error bank for the TCMs. The error bank helps protect the TCM error correction logic against livelocks, which can occur in the presence of hard errors.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bit[4] = 0b0 Includes hard error cache
```

The Cortex-R5 processor includes an error bank for the caches. The error bank helps protect the cache error correction logic against livelocks, which can occur in the presence of hard errors. This bank can be enabled or disabled using the system control registers.

```
MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
BIC r1, r1, #(0x1 << 22)     ; Enable hard-error support in the caches
MCR p15, 0, r1, c1, c0, 0    ; Write Secondary Auxiliary Control Register

...............

MRC p15, 0, r1, c1, c0, 0    ; Read Secondary Auxiliary Control Register
ORR r1, r1, #(0x1 << 22)     ; Disable hard-error support in the caches
MCR p15, 0, r1, c1, c0, 0    ; Write Secondary Auxiliary Control Register
```

The Cortex-R5 processor includes an event to indicate if the processor is in livelock due to hard errors. This event can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                  ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x62               ; Processor livelock
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

13.7.5 Bus protection on the Cortex-R5 processor

The Main AXI master interfaces, AXI peripheral ports, AHB peripheral ports, ACP ports and AXI slave port can be configured to include ECC generation and checking on the data buses, and parity generation and checking on the control buses. The TCM interfaces can be configured with parity generation and checking on the buses.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bit[10] = 0b1 Includes TCM Bus Parity
                           ; bit[3] = 0b1 Includes AMBA Bus ECC
```

The core will generate ECC or parity bits for output signals and will check ECC or parity bits for input signals.

If the core detects an error on one of the buses, it will signal the error with an event.

If the ECC checking detects a single bit ECC error, it will correct the error in-line in addition to signalling the error with an event.

These events can be captured and monitored by external hardware or the performance monitor unit for analysis.

```
MOV r0, #                  ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x70               ; Correctable Bus Fault
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB

.............................

MOV r0, #1                 ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x71               ; Fatal Bus Fault
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

To use the bus ECC and parity protection the system design around the core must also include the complementary parity and ECC bit generation and checking.

13.7.6 Redundant core in the Cortex-R5 processor

There is an implementation option in the Cortex-R5 processor to include a fully redundant copy of the processor implementation on the silicon. The outputs from the functional and redundant processors are compared with user defined logic to identify any differences in behavior. If a

difference is detected then this would be signalled with an event that might be sent to the processor or another logic block for analysis.

In the Cortex-R5 processor, the redundant core typically operates two cycles behind the primary core.

The exact behavior of the redundant core is implementation defined.

```
MRC p15, 0, r0, c15, c2, 1 ; read Build Options 2 Register
                           ; bit[31] = 0b1 Includes Redundant Core
```

13.7.7 Test of the fault detection and control features on the Cortex-R5 processor

There are two mechanisms to test that the ECC and parity features are working correctly in the Cortex-R5 processor

The AXI slave provides an interface to access the caches. Using this, it is possible to deliberately induce errors in the caches that can then be detected and corrected when accessed via instructions from the processor.

The AXI slave can be used to preload the TCMs but cannot be used to test errors in the TCMs.

The processor itself can be used to induce errors in the TCMs. ECC checking is enabled and disabled through the CP15. While ECC checking is disabled a sub-word access to a TCM location will not induce a read-modify-write process. This means that by doing sub-word accesses to TCM locations while ECC checking is disabled it is possible to generate a mismatch between the data value and the ECC code.

```
; Setup - these addresses MUST be doubleword aligned
LDR r10,=0x40000000     ; Address of victim TCM location to insert error
LDR r11,=0x40000008     ; Address of scratch memory location

LDR r0,=0xDEAFBEEF
LDR r1,=0xCAFEBABE

STR r0,[r10]
STR r1,[r10,#0x4]

; Disable ECC - ATCM
MRC p15,0,r0,c1,c0,1     ; Read aux ctl
BIC r0,r0,#0x02000000
MCR p15,0,r0,c1,c0,1


; Memory barrier to ensure all previous accesses have completed
DMB

; Read data and introduce ECC error
LDRD r0,[r10]            ; Read data to corrupt
STRD r0,[r11]            ; Store data to corrupt in scratch location

LSR  r2, r0, #16         ; Only interested in 3rd byte
EOR  r2, #0x2             ; Toggle one bit
```

```
STRB r2,[r11,#2]          ; Store byte to corrupt data and ECC
LDRD r0,[r11]             ; Load corrupted data back to set up
                          ; internal registers
BFI  r0, r1, #0, #8       ; Merge byte 0 from upper word into lower word
ROR  r0, r0, #24          ; Shift byte 3 into byte 0 position,
                          ; byte 0 into byte 1 position.
STRH r0, [r10,#3]         ; Store byte - data unchanged, but ECC changed

; Enable ECC
MRC p15,0,r0,c1,c0,1      ; Read aux ctl
ORR r0,r0,#0x02000000
MCR p15,0,r0,c1,c0,1
                          ; Read data - should get corrected from 0xDEAFBEEF
                          ; to 0xDEADBEEF

LDRD r0,[r10]
```

13.8 Fault detection and control features in the Cortex-R7 processor

Each of the Cortex-R series processors contain a number of features that enhance their suitability for safety-related applications. Semiconductor manufacturers may choose, or not choose, to implement these fault detection features in their processor implementation. These should not be assumed to be present in all Cortex-R devices

The Cortex-R7 processor provides the following optional fault detection features:

- ECC for Cache RAMs in the Cortex-R7 processor
- ECC for the TCMs on the Cortex-R7 processor
- BTAC and PRED RAM in the Cortex-R7 processor
- Hard error banks on the Cortex-R7 processor
- Bus protection on the Cortex-R7 processor
- Redundant core in the Cortex-R processors
- Test of the fault detection and control features on the Cortex-R7 processor

An additional feature of the Cortex-R7 processor is an MBIST interface for external analysis of errors. This cannot be used when the processor is running. However, it can be used when the processor is in `WFI` (Wait For Interrupt) state.

13.8.1 ECC for Cache RAMs in the Cortex-R7 processor

ECC generation and checking can be configured for the cache RAMs in the Cortex-R7 processor during implementation.

```
MRC p15, 0, r1, c1, c0, 1  ; Read ACTLR
ORR r1, r1, #(0x1 << 9)    ; Enable ECC detection on caches
MCR p15, 0, r1, c1, c0, 1  ; Write ACTLR
MRC p15, 0, r1, c1, c0, 1  ; Read ACTLR, bit[9] = 0, ECC not implemented
```

On detection of an ECC error in a clean cache line, the processor invalidates the line and then re-fetches the data from external memory.

On detection of a single-bit error in a dirty cache line the processor evicts the line, corrects it in-line, and then writes it back to main memory. The processor then invalidates the line and re-fetches the corrected data from external memory.

On detection of a double-bit error in a dirty cache line, the processor evicts the line and writes the data back to main memory. As the error cannot be corrected the words containing the double-bit error will not be written back to main memory. The processor invalidates the line and re-fetches the partially corrected data from external memory. This means that the instruction might use incorrect data.

When possible the location of the detected error is added to an Error bank, ready for analysis.

The ECC errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor Unit for analysis.

```
  MOV r0, #                 ; Select Counter 0
  MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
  ISB
  MOV r1, #0x65              ; Detected ECC errors on instruction cachet
  MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
  ISB

  ..............................

  MOV r0, #1                ; Select Counter 1
  MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
  ISB
  MOV r1, #0x66              ; Detected ECC errors on data cache
  MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
  ISB
```

The ECC events must also be used to signal to the processor, using an interrupt, that an error event occurred. The Interrupt handler must analyze the data in the Error banks to determine the location where the error occurred, whether the error is hard or soft, and what corrective or protective action must be taken.

```
  MRC p15, 0, r0, c15, c2, 0  ; Read DEER0, Data Cache Error Bank 0
  BIC r0, r0, #(0x1 << 0)     ; Clear Valid bit
  MCR p15, 0, r0, c15, c2, 0  ; Write DEER0, Data Cache Error Bank 0

.......................

  MRC p15, 0, r0, c15, c3, 2  ; Read IEER2, Instruction Cache Error Bank 2
  ORR r0, r0, #(0x1 << 1)     ; Set Hard Error bit
  MCR p15, 0, r0, c15, c3, 2  ; Write IEER2, Instruction Cache Error Bank 2

.........................
```

13.8.2 ECC for the TCMs on the Cortex-R7 processor

ECC generation and checking can be configured for the TCM RAMs on the Cortex-R7 processor during implmentation.

```
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR
ORR r1, r1, #(0x1 << 10)     ; Enable ECC detection on ITCM
ORR r1, r1, #(0x1 << 9)      ; Enable ECC detection on DTCM
MCR p15, 0, r1, c1, c0, 1    ; Write ACTLR
MRC p15, 0, r1, c1, c0, 1    ; Read ACTLR, bit[10:9] = 0b00 ECC not included
```

On detection of a single-bit error the data can be corrected and written back to the TCM. The core can then re-read the corrected data read from the same address.

On detection of a double-bit error the data cannot be corrected automatically. The core writes zeros to the corrupted data line.

When possible the location of the detected error is added to an Error bank, ready for analysis.

The ECC errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor unit for analysis.

```
MOV r0, #0                   ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5   ; Write PMNXSEL Register
ISB
MOV r1, #0x63                ; Detected ECC errors on ITCM
MCR p15, 0, r1, c9, c13, 1   ; Write EVTSELx Register
ISB

.............................

MOV r0, #1                   ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5   ; Write PMNXSEL Register
ISB
MOV r1, #0x64                ; Detected ECC errors on DTCM
MCR p15, 0, r1, c9, c13, 1   ; Write EVTSELx Register
ISB
```

The ECC events must also be used to signal to the core, using an interrupt, that an error event occurred. The Interrupt handler analyses the data in the Error banks to determine the location where the error occurred, whether the error is hard or soft, and what corrective or protective action must be taken.

```
MRC p15, 0, r0, c15, c4, 0   ; Read DTCMEER DTCM Error Bank
BIC r0, r0, #(0x1 << 0)      ; Clear Valid bit
MCR p15, 0, r0, c15, c4, 0   ; Write DTCMEER DTCM Error Bank

.......................

MRC p15, 0, r0, c15, c5, 0   ; Read ITCMEER ITCM Error Bank
ORR r0, r0, #(0x1 << 1)      ; Set Hard Error bit
MCR p15, 0, r0, c15, c5, 0   ; Write ITCMEER ITCM Error Bank
```

13.8.3 BTAC and PRED RAM in the Cortex-R7 processor

Parity bit generation and checking can be included for the BTAC and Prediction RAM in the Cortex-R7 processor.

```
MRC p15, 0, r1, c1, c0, 1  ; Read ACTLR
ORR r1, r1, #(0x1 << 9)    ; Enable ECC detection on caches
MCR p15, 0, r1, c1, c0, 1  ; Write ACTLR
MRC p15, 0, r1, c1, c0, 1  ; Read ACTLR, bit[9] = 0b0,
                           ; PRED and BTAC parity not implemented
```

Parity errors are signalled with events. These events can be captured and monitored by external hardware or the Performance Monitor Unit and for analysis.

```
MOV r0, #0                 ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x61              ; Parity error on PRED
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB

.............................

MOV r0, #1                 ; Select Counter 1
MCR p15, 0, r0, c9, c12, 5 ; Write PMNXSEL Register
ISB
MOV r1, #0x62              ; Parity error on BTAC
MCR p15, 0, r1, c9, c13, 1 ; Write EVTSELx Register
ISB
```

No corrective action is taken by the processor on detection of a parity error in the BTAC or Prediction RAMs. An error in these RAMs can generate a mis-prediction in the instruction prefetch logic, this can have an impact on processor performance but will not affect the functionality of the core.

13.8.4 Hard error banks on the Cortex-R7 processor

The Cortex-R7 processor includes hard error banks for the caches and TCMs. The error bank helps protect the cache and TCM error correction logic against livelocks, which can occur in the presence of hard errors.

The Cortex-R7 processor provides a bank of eight registers for errors:

- 3 for the instruction cache.
- 3 for the data cache.
- 1 for ITCM.
- 1 for DTCM.

The error registers contain the location of the error and specify whether it is a hard error or not. These registers are automatically filled with the error location whenever an error is detected. However, whether it is a hard error or not is filled in by the core software instead of the hardware.

These error registers can be accessed using the MCR and MRC instructions when the core is in privileged mode.

```
MRC p15, 0, r0, c15, c2, 0  ; Read DEER0, Data Cache Error Bank 0
MCR p15, 0, r0, c15, c2, 0  ; Write DEER0, Data Cache Error Bank 0

MRC p15, 0, r0, c15, c2, 1  ; Read DEER1, Data Cache Error Bank 1
MCR p15, 0, r0, c15, c2, 1  ; Write DEER1, Data Cache Error Bank 1

MRC p15, 0, r0, c15, c2, 2  ; Read DEER2, Data Cache Error Bank 2
MCR p15, 0, r0, c15, c2, 2  ; Write DEER2, Data Cache Error Bank 2

MRC p15, 0, r0, c15, c3, 0  ; Read IEER0, Instruction Cache Error Bank 0
MCR p15, 0, r0, c15, c3, 0  ; Write IEER0, Instruction Cache Error Bank 0

MRC p15, 0, r0, c15, c3, 1  ; Read IEER1, Instruction Cache Error Bank 1
MCR p15, 0, r0, c15, c3, 1  ; Write IEER1, Instruction Cache Error Bank 1

MRC p15, 0, r0, c15, c3, 2  ; Read IEER2, Instruction Cache Error Bank 3
MCR p15, 0, r0, c15, c3, 2  ; Write IEER2, Instruction Cache Error Bank 3

MRC p15, 0, r0, c15, c4, 0  ; Read DTCMEER DTCM Error Bank
MCR p15, 0, r0, c15, c4, 0  ; Write DTCMEER DTCM Error Bank

MRC p15, 0, r0, c15, c5, 0  ; Read ITCMEER ITCM Error Bank
MCR p15, 0, r0, c15, c5, 0  ; Write ITCMEER ITCM Error Bank
```

13.8.5 Bus protection on the Cortex-R7 processor

Parity and ECC generation and checking can be configured for some of the Cortex-R7 processor AXI bus signals. The Main AXI master interfaces, AXI peripheral port and ACP port can be configured to include ECC generation and checking on the data buses, and parity generation and checking on the control buses.


Note

There is no register in the Cortex-R7 processor that indicates whether bus protection has been included in the configuration. However, bus protection can only be included if the RAM ECC has also been included.

The Cortex-R7 processor will generate ECC or parity bits for output signals and will check ECC or parity bits for input signals.

If the core detects an error on one of the buses it will signal the error with an event.

If the ECC checking detects a single bit ECC error it will correct the error in-line and will signal the error with an event.

```
MOV r0, #0                  ; Select Counter 0
MCR p15, 0, r0, c9, c12, 5  ; Write PMNXSEL Register
ISB
MOV r1, #0x6A                ; Correctable ECC error on master 0 bus
MCR p15, 0, r1, c9, c13, 1  ; Write EVTSELx Register
ISB
```

To use the bus ECC and parity protection, the system design around the core must also include the complementary parity and ECC bit generation and checking.

13.8.6 Redundant core in the Cortex-R processors

There is an implementation option in the Cortex-R7 processor to include a fully redundant copy of the core implementation on the silicon. The outputs from the functional and redundant core are compared with user defined logic to identify any differences in behavior. If a difference is detected then this would be signalled with an event that might be sent to the core or another logic block for analysis.

In the Cortex-R7 processor, the redundant core typically operates 2 cycles behind the primary core.

The exact behavior of the redundant core comparison logic is implementation defined.

There is no register in the Cortex-R7 processor that indicates whether a redundant core has been included in the configuration.

13.8.7 Test of the fault detection and control features on the Cortex-R7 processor

Unlike the Cortex-R5 and Cortex-R4 processors, the Cortex-R7 processor provides a method for writing to the TCMs and caches using system control registers:

Cache and TCM Debug Operation Register (CTDOR)

This is used to detail and trigger the required access (RAM selection, address/index, read/write).

RAM Access Data Register (RADRLO and RADRHI)

This stores data values for the RAM operation.

RADRLO is used for the 32 bit data RAM accesses.

RADRLO and RADRHI are used for the 64 instruction RAM accesses.

RAM Access ECC Regisrer (RAECCR)

This stores the ECC value for the CTDOR RAM operation.

With these registers it is possible to introduce errors easily because data stored in the TCM or cache can be changed without updating the ECC. It is also possible to change the ECC without changing the data in the TCM. Using this feature it is possible to deliberately induce errors in a TCM or cache that can then be detected and corrected when accessed by instructions from the processor.

```
; Modify Data stored in the Data Cache, Way 2, Index 4, Word 3

; First read the cache location using CTDOR
MOV r1, #0
```

```
  ORR r1, r1, #2<<30          ; Way selection
  ORR r1, r1, #1<<22          ; Cache/TCM RAM selection, select Cache
  ORR r1, r1, #1<<21          ; Data/Tag RAM selection, select Data
  ORR r1, r1, #0<<20          ; Instruction/Data selection, select Data
  ORR r1, r1, #4<<5           ; Index selection
  ORR r1, r1, #3<<2           ; Word selection
  ORR r1, r1, #0<<0           ; Read operation

  MCR p15, 0, r1, c15, c1, 0  ; Write CTDOR with operation selected above

  ; The CTDOR access updates the value in the low RAM Access Data Register
  ; and RAM Access ECC Register

  MRC p15, 0, r3, c15, c1, 1  ; Read data from low RAM Access Data Register
  MOV r2, #0
  ORR r2, r2, #1<<5
  EOR r3, r2,                 ; Deliberately corrupt bit 5 in read word
  MRC p15, 0, r3, c15, c1, 1  ; Copy corrupted data to CP15 register

  ORR r1, r1, #1<<0           ; Select CTDOR Write operation
  MCR p15, 0, r1, c15, c1, 0  ; Update CTDOR and trigger write of
                              ; corrupted data to data cache

..............................

  ; Modify ECC code stored for the Instruction Cache, Way 0, Index 3,
  ; Words 6 and 7.

  ; First read the cache location using CTDOR
  MOV r1, #0
  ORR r1, r1, #2<<30          ; Way selection
  ORR r1, r1, #1<<22          ; Cache/TCM RAM selection, select Cache
  ORR r1, r1, #1<<21          ; Data/Tag RAM selection, select Data
  ORR r1, r1, #1<<20          ; Instruction/Data selection, select Instruction
  ORR r1, r1, #3<<5           ; Index selection
  ORR r1, r1, #6<<2           ; Word selection
  ORR r1, r1, #0<<0           ; Read operation

  MCR p15, 0, r1, c15, c1, 0  ; Write CTDOR with operation selected above

  MRC p15, 0, r4, c15, c1, 3  ; Read ECC chunk from RAM Access ECC Register
  MOV r2, #0
  ORR r2, r2, #1<<2
  EOR r3, r2                  ; Deliberately corrupt bit 2 in ECC word
  MRC p15, 0, r4, c15, c1, 3  ; Copy corrupted ECC chunk to CP15 register
                              ; (useless in this case)

  ORR r1, r1, #1<<0           ; Select CTDOR Write operation
  MCR p15, 0, r1, c15, c1, 0  ; Update CTDOR and trigger write of
                              ; corrupted ECC to instruction cache
```

14. Profiling

Profiling is a technique that lets you identify sections of code that consume large proportions of the total execution time. It is usually more productive to focus optimization efforts on code segments that are executed very frequently, or that take a significant proportion of total execution time than to optimize rarely used functions or code that takes only a small proportion of total execution time. A profiler will tell you which parts of the code are frequently executed and which occupy the most core cycles. A profiler can help you identify bottlenecks, situations where the performance of the system is constrained by a small number of functions. This data is collected using instrumentation, an execution trace or sampling.

When you have identified some slow part of your code it is important to consider whether you can change the algorithm, before attempting to improve the existing code. For example, if the time is being spent searching a linked list, it is probably much more beneficial to change to using a tree or hash table instead of spending effort to speed up the linked list search.

Profiling can be considered as a form of dynamic code analysis. Profiling tools can gather information in a number of different ways. There are two basic approaches to gathering information:

Time based sampling

The state of the system is sampled at a periodic, time-based interval. The size of this interval can affect the results. A smaller sampling interval can increase execution time but produce more detailed data.

Event based sampling

Sampling is driven by occurrences of an event, which means that the time between sampling intervals is usually variable. Events can often be hardware related, for example, cache misses.

It is also important to understand that profilers typically operate on a statistical basis. They might not necessarily produce absolute counts of events. In complex systems, it might be necessary to control profiling information by using annotation options to specify:

- Which events are to be recorded.
- Which events are to be shown.
- Thresholds to avoid displaying large numbers of functions with low count numbers.

14.1 Profiler output

Profiler tools normally provide two kinds of information:

Call graph

The call graph tells you the number of times each function was called. This can help point out which function calls can be eliminated or replaced and shows inter-relations between different functions. Viewing a call graph can suggest code to optimize and reveal hidden

bugs, for example, if code is unexpectedly calling an error function many times. Collecting call graph information can require building the code with special options.

Flat profile

A flat profile, as in the example below, shows how much core time each function uses and the number of times it was called. This enables a simple identification of which functions consume large fractions of run-time and should therefore be considered first for possible optimizations.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name

 33.34     0.02     0.02     6275     0.00     0.00  start
 16.67     0.03     0.01      192     0.07     0.21  func1
 16.67     0.04     0.01       15     1.20     1.20  memcpy
 16.67     0.05     0.01        7     1.41     1.41  write
```

14.2 Performance Monitor Unit

The Cortex-R series processors include a Performance Monitor Unit (PMU). The PMU is a powerful profiling feature that measures and analyzes the processor performance.

The PMU hardware is able to count several events, using multiple counters. Combining values from different counters provides useful parameters to optimize. For example, if the PMU counts the total number of clock cycles and the number of instructions executed, then these can be used to calculate the cycles per instruction. The cycles per instruction is a useful proxy for the efficiency with which the core is operating. It is possible to generate information about cache hit or miss rates, separately for both L1 data and instruction caches. It is possible to examine how code changes can affect these parameters.

The Cortex-R series processors contain event counting hardware which can be used to:

- Profile and benchmark code.
- Generate cycle and instruction count figures.
- Derive figures for cache misses and other parameters.

The performance counter block contains a cycle counter which can count core cycles, or be configured to count every 64 cycles. There are also a number of configurable 32-bit wide event counters which can be set to count instances of events from a wide-ranging list, for example, instructions executed, or exceptions taken.

These counters can be accessed through debug tools, or by software running on the core, through the CP15 PMU registers. They provide a non-invasive debug feature and do not change the behavior of the core. CP15 also provides a number of controls for enabling and resetting the counters, and to indicate overflows. There is an option to generate an interrupt on a counter overflow. The cycle counter can be enabled independently of the event counters.

The PMU registers are accessible in privileged modes. You can use the User Enable (PMUSERENR) Register to make all the PMU registers, except for the Interrupt Enable Set (PMINTENSET) and Interrupt Enable Clear (PMINTENCLR) Registers, accessible in User mode.

It is important to understand that information generated by such counters might not be exact. In a superscalar, out-of-order processor, for example, it can be difficult to guarantee that the number of instructions executed is precise at the time any other counter is updated.

The following table lists the standard countable events common to all ARMv7-R based processors. There are additional events that can be monitored. For more information, see the Technical Reference Manual for the processor.

Table 14-1: Performance monitor events

Number	Event counted
0x00	Software increment of the Software Increment Register
0x01	Instruction fetch that causes a Level 1 instruction cache refill
0x03	Data fetch that causes a Level 1 data cache refill
0x04	Data Read or Write operation that causes a Level 1 data cache access
0x06	Memory-reading instruction executed
0x07	Memory-writing instruction executed
0x08	Instruction architecturally executed
0x09	Exception taken
0x0A	Exception return executed
0x0B	Instruction that writes to the Context ID register
0x0C	Software change of program counter
0x0D	Immediate branch instruction executed
0x0E	Procedure return, other than exception return
0x0F	Unaligned load or store
0x10	Branch mispredicted or not predicted
0x11	Cycle count; the register is incremented on every cycle
0x12	Predictable branch speculatively executed

15. Coding for Cortex-R Processors

You can optimize code for power, speed, code density or memory footprint. There are many GNU GCC and ARM Compiler features that take advantage of the Cortex-R series design to generate optimized code.

15.1 Compiler optimizations

The ARM Compiler and GNU GCC provide a wide range of options to increase the speed, or reduce the size, of the executable files they generate. For each line in the source code, there are generally many possible choices of assembly instructions that could be used. The compiler must trade-off a number of resources, such as registers, stack and heap space, code size, number of instructions, compilation time, ease of debug, and number of cycles per instruction to produce an optimized image file.

15.1.1 Idiom recognition

The ARM and Thumb instructions sets include a number of specialist instructions that do not map easily to C operations. These can be system control instructions, such as enabling or disabling interrupts, or specialist data processing instructions.

An example of a specialist data processing instruction would be the `REV` instruction. This instruction reverses the order of bytes in a register, changing the endian format. Because the concept of endianness is not understood by C directly, it is difficult to express in C that this is what you want to do. The compiler attempts to recognize common code fragments that implement such behavior, and use the specialist instructions. This is known as idiom recognition. The following shows code compiled using RVCT 4.0 build 870, with `-O2` optimization level:

C code:

```
uint32_t rev (uint32_t a)
{
return
((a<<24) & 0xFF000000U) |
((a<< 8) & 0x00FF0000U) |
((a>> 8) & 0x0000FF00U) |
((a>>24) & 0x000000FFU);
}
```

Assembly produced with `--cpu=v5TE`:

```
rev PROC
 MOV  r1,#0xff0000
 AND  r1,r1,r0,LSL #8
 MOV  r2,#0xff00
 ORR  r1,r1,r0,LSL #24
 AND  r2,r2,r0,LSR #8
```

```
 ORR  r1,r1,r2
 ORR  r0,r1,r0,LSR #24
 BX   lr
ENDP
```

Assembly produced with `--cpu=Cortex-R4F`:

```
rev PROC
 REV  r0,r0
 BX   lr
ENDP
```

The compiler might not recognize all cases where a specialist instruction is required. In these cases the code can be re-written in assembler or using compiler intrinsics. An intrinsic tells the compiler to use a specific instruction. For example:

```
 __usat()
```

This kind of optimization is quite intensive in terms of a developer's time. Typically such effort can only be justified for frequently used code, or code that forms part of a critical section of the application. The PMU can be used to help identify such critical sections.

15.1.2 Function inlining

When a function is called, there is a certain overhead. A called function must store its own return address on the stack if it has to reuse `R14`. Instructions might also be required to place arguments in the appropriate registers and push registers on the stack, in accordance with the Procedure Call Standard. There is a possible overhead when returning to the original point of execution when the function ends, again requiring a branch (and corresponding instruction pipeline flush) and possibly popping registers from the stack. However, the pipeline will not be flushed if the return is correctly predicted using the return stack. This function-call overhead can become significant when there are functions that contain only a few instructions, and where these functions represent a significant amount of the total run-time. Also, executing branches uses branch predictor resources, that can affect overall program performance. Function inlining eliminates this overhead by replacing calls to a function by a copy of the actual code of the function itself (known as placing the code inline).

Inlining for critical code paths is always a worthwhile optimization if there is only one place where the function is called. It is always worthwhile if calling the function requires more instructions (memory) than inlining the function body. An additional consideration is that inlining can help permit other optimizations. Clearly, increasing the number of times that a function is called will increase the number of inlined copies of the function that are made and this will increase the cost in code size.

GCC performs inlining only within each compilation unit. The `inline` keyword can be used to request that a specific function must be inlined wherever possible, even in other files. The GCC documentation gives more details of this and how its use can be combined with `static` and `extern`.

We will look at inlining in a little more detail when we consider cache optimizations.

15.1.3 Eliminating common sub-expressions

Another simple source-level optimization is re-using already computed results in a later expression. This common sub-expression elimination is performed automatically when optimization command line switches are used and can make code both smaller and faster. However, the compiler might not necessarily catch all cases, and it can sometimes be more useful to do this by hand.

The following example illustrates how this works:

```
i = a * b + c;
j = a * b * d;
```

The compiler can treat this code as if it had been written as follows. It must be noted though, that it can only do this if neither `a` nor `b` is volatile.

```
tmp = a * b;
i = tmp + c;
j = tmp * d;
```

This reduces both the instruction count and cycle count.

15.1.4 Loop unrolling

Every iteration of a loop has a certain penalty associated with it. Every conditional loop must include a test for the end of loop on each iteration. Additionally, there is a branch instruction to iterate over the loop, that can take a number of cycles to execute. You can avoid this penalty by unrolling loops, partially or fully.

Consider the simple code shown below, to initialize an array.

```
for (i = 0; i < 10; i++)
{
   x[i] = i;
}
```

Each iteration of the loop contains an assembler sequence of the form shown below:

```
CMP i,#10
BLT for_loop
```

A large proportion of the total run time will have been spent checking if the loop has terminated and in executing a branch to re-execute the loop.

The same code can be written by unrolling the loop, as shown below:

```
x[0] = 0;
x[1] = 1;
x[2] = 2;
x[3] = 3;
x[4] = 4;
x[5] = 5;
x[6] = 6;
x[7] = 7;
x[8] = 8;
x[9] = 9;
```

When the code is written in this way, you remove the compare and branch instruction and have a sequence of stores and adds. This is clearly larger than the original code but can execute considerably faster.

Conventionally, loop unrolling is often considered to increase the speed of the program but at the expense of an increase in code size (except for very short loops). However, in practice this might not always be the case on many hardware platforms. In many systems, an access to external memory takes significant numbers of cycles and an instruction cache is provided. Code that loops will often fit into the cache very well. The code is fetched into the cache during the first loop iteration and is executed directly from cache after that. Unrolling the loop can mean that the code is executed only once and, because it is larger, does not cache so well. This is more likely to be the case for functions that are executed only once. Loops that are executed frequently might be cached whether they are unrolled or not. An additional consideration is that modern ARM processors typically include branch prediction logic that can hide the effect of pipeline flushes from you by speculatively predicting whether a branch will or will not be taken ahead of the actual evaluation of a condition. In some cases, the branch instruction can be folded, so that it does not require an actual processor cycle to execute.

Cortex-R series processors can have long, complex instruction pipelines, with interdependencies between instructions, particularly loads and instructions that set condition code flags. The compiler understands the rules associated with a particular processor and can often re-arrange instructions so that pipeline interlocks are avoided. This is called scheduling and typically involves re-arranging the order of instructions in ways that do not alter the logical correctness of the program or its size, but that reduce its execution time. This can significantly increase the compiler effort, increasing both the time and memory required for the compilation. It can also restrict the ability to perform source level debug. There might no longer be a strict one-to-one link between a line of C source and a sequence of assembly instructions. You can instead have a couple of instructions from a C statement followed by instructions for the next statement and then some more instructions for the first statement.

15.1.5 GCC optimization options

GCC has a range of optimization levels, plus individual options to enable or disable particular optimizations.

The overall compiler optimization level is controlled by the command line option, where `n` is the required optimization level, as follows:

- `-O0`. (default). No optimization is performed. Each source code command relates directly to the corresponding instructions in the executable file. This gives the clearest view for source level debugging.
- `-O1`. This enables most common forms of optimization that requires no size versus speed decisions, including function inlining. It can often actually produce a faster compile than -O0, because the resulting files are smaller.
- `-O2`. This enables additional optimizations, such as instruction scheduling. Again, optimizations that can have speed versus size implications will not be used.
- `-O3`. This enables additional optimizations, such as aggressive function inlining and can therefore increase the speed at the expense of image size.
- `-funroll-loops`. This option is independent of the `-On` option, and enables loop unrolling. Loop unrolling can increase code size and might not have a beneficial effect in all cases.
- `-Os`. This selects optimizations that attempt to minimize the size of the image, even at the expense of speed.

Higher levels of optimization can restrict debug visibility and increase compile times. It is usual to use `-O0` for debugging, and `-O2` for finished code. When using these optimization options with the `-g` (debug) switch, it can be difficult to see what is happening. The optimizations can change the order of statements or remove (or add) temporary variables among other things. But an understanding of the kinds of things the compiler will do means that satisfactory debug is normally still possible with `-O2 -g`.

For optimal code, it is important to specify to the compiler as much detailed information about the target platform as practically possible. Many useful options are documented on http://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html.

The main platform-specifying parameters are:

```
-march=<arch>
```

where `<arch>` is the architecture version to compile for. This defines the instruction set supported, It can make a significant difference to performance to specify `-march=armv7-a` if this is supported by your platform but is not used by default by your compiler.

```
-mcpu=<cpu>
```

More specific than `-march`, `-mcpu` specifies which processor to optimize for, including scheduling instructions in the way most efficient for that processor's pipeline.

```
-mtune=<cpu>
```

This option provides processor specific tuning options for code, even when only an architecture version is specified on the command line. For instance, the command line might contain -

`march=armv5te -mtune=cortex-r7`. This selects instructions for the architecture ARMv5TE but tunes the selected instructions for execution on the Cortex-R7 processor.

```
-mfpu=<fpu>
```

If your target platform supports hardware floating-point, specify this to ensure that the compiler can make use of these instructions. For a Cortex-R5F target, you would specify `-mfpu=vfpv3-d16`.

```
-mfloat-abi=<name>
```

This option specifies the floating-point ABI to use. Values for `<name>` are:

`soft`
causes GCC to generate code containing calls to the software floating-point library for floating-point operations.

`softfp`
enables GCC to generate code containing hardware floating-point instructions, but still uses the software floating-point linkage.

`hard`
enables GCC to generate code containing hardware floating-point instructions and uses FPU-specific hardware floating-point linkage.

The default depends on the target configuration. You must compile your entire program with the same ABI, and link with a compatible set of libraries.

Table 15-1: Floating-point code generation on page 185 shows a few examples of code generation for floating-point operations.

Table 15-1: Floating-point code generation

-mfpu	-mfloat-abi	Resultant code
Any value	`soft`	Floating-point emulation using software floating-point library
`vfpv3`	`softfp`	VFPv3 floating-point code
`vfpv3-d16`	`softfp`	VFPv3 floating-point code

15.1.6 armcc optimization options

The armcc compiler enables you to compile your C and C++ code. It is an optimizing compiler with a range of command-line options to enable you to control the level of optimization.

The command line option gives a choice of optimization levels, as follows:

- `-Ospace`. This option instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time.
- `-Otime`. This option instructs the compiler to perform optimizations to reduce execution time at the expense of a possible increase in image size.

- `-O0`. Turns off most optimizations. It gives the best possible debug view and the lowest level of optimization.
- `-O1`. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with `--debug`, this option gives a satisfactory debug view with good code density.
- `-O2` (default). High optimization. If used with `--debug`, the debug view might be less satisfactory because the mapping of object code to source code is not always clear.
- `-O3`. performs the same optimizations as `-O2` however the balance between space and time optimizations in the generated code is more heavily weighted towards space or time compared with `-O2`. That is:
  - `-O3 -Otime` aims to produce faster code than `-O2 -Otime`, at the risk of increasing your image size
  - `-O3 -Ospace` aims to produce smaller code than `-O2 -Ospace`, but performance might be degraded.

15.2 Endianness

There are two basic ways of viewing bytes in memory - little-endian and big-endian. On big-endian machines, the most significant byte of an object in memory is stored at the least significant (closest to zero) address. On little-endian machines, the most significant byte is stored at the highest address.

The term byte-ordering can also be used rather than endian. Other kinds of endianness do exist, notably middle-endian and bit-endian, but we will not describe these.

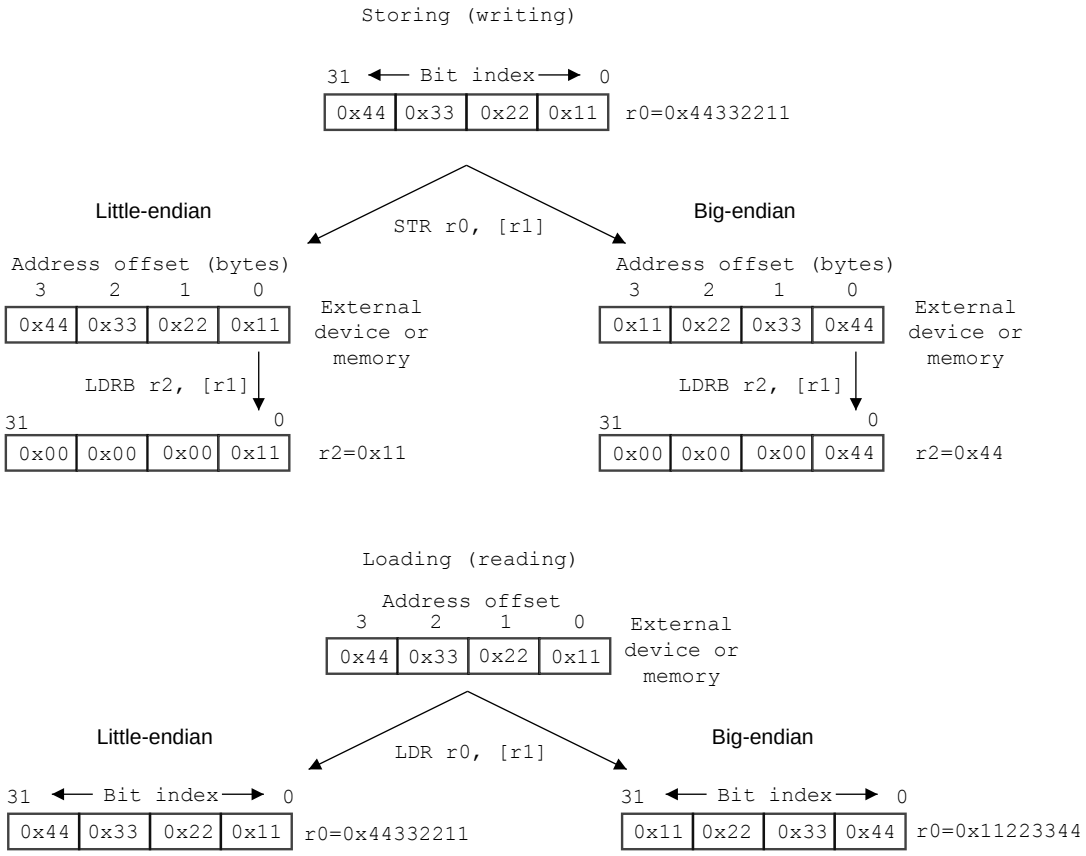
Consider the following simple piece of code:

```
int i = 0x44332211;
unsigned char c = *(unsigned char *)&i;
```

On a 32-bit big-endian machine, `c` is given the value of the most significant byte of `i`: `0x44`. On little-endian machines, `c` is the least significant byte of `i`: `0x11`.

Figure 15-1: Different endian behaviors on page 187 illustrates the two differing views of memory. It should be stated at this point that many people find endianness confusing and that even the act of drawing a diagram to illustrate it can reveal a personal bias. The diagram shows a 32-bit value in a register being written to address `0x1000`, using a `STR` instruction. The core then performs a read of a byte, using a `LDRB` instruction. A different value will be returned by this instruction sequence depending on whether you have a little- or big-endian memory system.

Figure 15-1: Different endian behaviors



ARM cores support both modes, but are most commonly used in, and typically default to little-endian mode. Most Linux distributions for ARM tend to be little-endian only. The x86 architecture is little-endian.

So, there are two issues to consider, code portability and data sharing. Systems are built from multiple blocks and can include one or more cores, DSPs, peripherals, memory, and network connections. Whenever data is shared between these elements, there is a potential endianness conflict. If code is being ported from a system with one endianness to a system with different endianness, it might be necessary to modify that code, either to make it endian-neutral or to work with the opposite byte-ordering.

Cortex-R series processors provide support for systems of either endian configuration, controlled by the CPSR E bit that enables software to switch dynamically between viewing data as little or big-endian. Instructions in memory are always treated as little-endian. The REV instruction (see Byte reversal) can be used to reverse bytes within an ARM register, providing simple conversion between big and little-endian formats.


Note

The Cortex-R4 and Cortex-R5 can be configured to support big-endian Instruction code. This is to support legacy systems.

In principle, it is straightforward to support mixed endian systems. Typically this means the system is natively of one endian configuration, but there are peripherals which are of the opposite endianness. The CPSR E bit can be modified dynamically by software, and there is a `SETEND` instruction provided to do this. The CP15:SCTLR (System Control Register, c1), contains the EE bit (see Coprocessor 15) that defines the endian mode to switch to on an exception.

Modern ARM processors support a big-endian format known architecturally as BE-8 that is only applied to the data memory system. Older ARM processors used a different format known as BE-32 that applied to both instructions and data. BE-8 corresponds to what most other computer architectures call big-endian.

The following example provides a simple piece of code that behaves differently when run on architectures with different endianness.

```
int i- 0x12345678;
char *buf = (char*)&i;
char i0, i1, i2, i3;

i0 = buf[0];
i1 = buf[1];
i2 = buf[2];
i3 = buf[3];
```

The values of `i0...i3` are not guaranteed to be the same if the system endianness changes. This kind of code is therefore inherently non-portable.

When inspecting code in which you suspect endianness problems, you must look for the following potential causes of problems:

Unions

A union can hold objects of different types and sizes. You must keep track of what the data member represents at any particular time. Code that uses unions must be carefully checked. If the union is used to access the same data, but with different data types, there exists a possible endianness, alignment, and packing problem. Any time that halfword, word (or longer) data types are combined or viewed as an array of bytes is a potential issue.

Casting of data types

Anywhere that data is accessed in a way outside of its native data type is a potential problem. Similarly, if there are arrays of bytes, they must not be accessed other than as a byte data type. Casting of pointers changes how data is addressed and can be endian sensitive.

Bitfields

To avoid endianness problems code that defines bitfields or performs bit operations must not be used in code that is intended to be portable.

Data sharing

Any code that reads shared data from another block, or exports data to another block, must be checked to see whether the two blocks agree endian definitions. If the two are different, it might be necessary to implement byte swapping at one location.

Network code

Code that accesses networking or other I/O devices must be reviewed to see if there is any endian dependency. Again, it might be necessary to re-write code for greater efficiency, or swap bytes at the interface.

15.3 ARM memory system optimizations

Writing code that is optimal for the system it will run on is a key part of the art of programming. It requires you to understand how the compiler and underlying hardware will carry out the tasks described in the lines of code. If the processor performs the tasks on-chip, with fewer access to external memory, it reduces the power consumption. Additionally, by accessing the external memory less frequently, it improves the system performance, enabling software to run faster. This also means that the processor can be clocked slower, to save power.

15.3.1 Use of cache

The Cortex-R series processors are optimized for execution from the caches or TCM. Operating from non-cached external memory will often have a significant impact.

Performance is also affected by different caching strategies, for example:

- read-allocate or write-allocate
- write-through or write-back.

Write-back, write allocate is often a good choice for stack space. Data is written first (pushed) and then read back (popped), that fits a write-allocate strategy. Because the memory is also frequently re-written, write-back prevents unnecessary updates of memory. However, when using parity checking on the caches write-back is not supported.

Code should be structured in a way that ensures maximum re-use of data already loaded into the cache. It is this principle of data locality, the degree to which accesses to the same cache line are concentrated during program execution, in both space and time, that gives best performance.

15.3.2 Loop tiling

Loop tiling divides loop iterations into smaller pieces, in a way which promotes data cache re-use. Large arrays are divided into smaller blocks (tiles) that match the accessed array elements to the cache size. The classic example to illustrate this approach is a large matrix vector product.

Consider two square matrices `a` and `b`, each of size 1024 × 1024. The following example shows code to compute a matrix vector product. This requires you to multiply each element in each array with each element in the other array.

```
for (i = 0; i < 1024; i++)
  for (j = 0; j < 1024; j++)
    for (k = 0; k < 1024; k++)
      result[i][j] = result[i][j] + a[i][k] * b[k][j];
```

In this case, the contents of matrix `a` are accessed sequentially, but matrix `b` advances in the inner loop, by row. It is therefore, highly probable that you will encounter a cache miss for each multiply operation.

It is obvious that the order in which the additions for each element of the result matrix are calculated does not change the result, ignoring the effect of such things as overflows. Code can be rewritten in a way that improves the cache hit rate. In the example, the elements of matrix b are accessed in the following way (0,0), (1,0), (2,0)... (1023, 0), (0,1), (1,1)... (1023,1). The elements are stored in memory in the order (0,0), (0,1) etc. For word sized elements, it means that the elements (0,0), (0,1)...(0,7) is stored in the same cache line. For simplicity, we will assume that the start address of the matrix is aligned to a cache line. Alignment will be mentioned again in Structure alignment. Therefore, elements (0,0), (0,1), (0,2), ... will be in the same cache line; when you load (0,0) into the cache, you get (0,1...7) too. By the time the inner loop completes, it is likely that this cache line will be evicted.

If you modify the code so that two (or indeed four, or eight) iterations of the middle loop are performed immediately while executing the inner loop, as in the following example, you can make a big improvement. Similarly, you can unroll the outer loop two (or four, or eight) times as well.

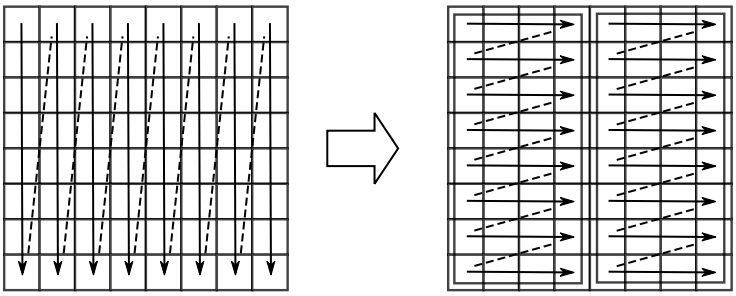
```
for (io = 0; io < 1024; io += 8)
  for (jo = 0; jo < 1024; jo += 8)
    for (ko = 0; ko < 1024; ko += 8)
      for (ii = 0, rresult = &result[io][jo],
        ra = &a[io][ko]; ii < 8;
        ii++, rresult += 1024, ra += 1024)
        for (ki = 0, rb = &b[ko][jo];
          ki < 8; ki++, rb += 1024)
          for (ji = 0; ji < 8; ji++)
            rresult[ji] += ra[ki] * rb[ji];
```

There are now six nested loops. The outer loops iterate with steps of 8, representing the fact that eight `int` sized elements are stored in each line of the level 1 cache. Some additional optimizations have also been introduced. The order of `ji` and `ki` has been reversed as only one expression uses `ki`, but two use `ji`. In addition, you can optimize by removing common expressions from the inner loops. All pointer accesses are potential sources of aliasing in C, so by using `result`, `ra` and `rb` to

access array elements, the array indexing is speeded up. This is covered in more detail in Source code modifications.

Figure ¿fig:xxx? illustrates the changing cache access pattern that results from changes to the C code.

Figure 15-2: Effect of tiling on cache usage



15.3.3 Loop interchange

In many programs, there are nested loops. Avery simple example would be code that stepped through the items in a 2-dimensional array. For reasonably complex code, you can sometimes get better performance by re-arrangement of the loops. It is better to have the loop with the smaller number of iterations as the outer loop and the one with the highest iteration count as the innermost loop.

This gives two potential advantages. One is that the compiler can potentially unroll the inner loop. Perhaps more importantly for complex loops where the size of the nested loop is sufficiently large that it might not all be held in the level 1 cache at the same time, the overall cache hit rate is improved by this change. Some compilers can make this change automatically at higher levels of optimization. For example, GCC 4.4 adds the switch `-floop-interchange` to do this.

15.3.4 Structure alignment

Efficient placement of structure elements and alignment are not the only aspects of data structures that influence cache efficiency. Where code has a large working set, it is important to make efficient use of the available cache space. To achieve this, it might be necessary to rearrange data structures.

It is common to have data structures that span multiple cache lines, but where the program uses only a few parts of the structure at any particular time. If there are many objects of this type, it can make sense to try to split the structure so that it fits within a cache line. For example, you can split an array of structures into two or more arrays of smaller structures. This only makes sense if the object itself is aligned to a cache boundary. For example, consider the case where you have a very large array of instances of a 64-byte structure (much larger than the cache size). Within that structure, you have a byte-sized quantity and you have a commonly used function that iterates through the array looking only at that byte-sized quantity. This function would make inefficient use

of the cache, as you would have to load an entire cache line to read the 8-bit value. If instead those 8-bit values were stored in their own array (rather than as part of a larger structure), you would get 32 or 64 values per cache linefill.

Unaligned accesses are supported, but can take extra cycles in comparison to aligned accesses. For performance reasons, therefore, it can be sensible to remove or reduce unaligned accesses.

15.3.5 Associativity effects

As we have seen, ARM L1 caches are normally 4-way set-associative, but L2 caches typically have 8- or 16-way associativity. There can be performance problems if more than four of the locations in the data fall into the same cache set, as there can be repeated cache misses, even though other parts of the cache can be unused. The ARM L1 Cache uses physical rather than virtual addresses, so it can be difficult for programmers operating in User mode to take care of this.

A particularly common cause of this problem is arranging data so that it is on boundaries of powers of two. If the cache size is 16KB, each way is 4KB in size. If you have multiple blocks of data arranged on boundaries that are multiples of 4KB, the first access to each block will go into line 0 of a way. If code accesses the first line in several such blocks then you can get cache misses even if only five cache lines in total are being used. Unaligned accesses can increase the likelihood of this, as each access might require two cache lines rather than one.

15.3.6 Optimizing instruction cache usage

The C programmer does not directly have control over how the instruction cache is used by code. Code is linear between branch instructions and this pattern of sequential accesses uses the cache efficiently. The branch prediction logic of the core will try to minimize the stalls because of branches, so there is little you can do to assist. The main goal for you is to reduce the code footprint. Many of the compiler optimizations enabled at `-O2` and `-O3` for the ARM Compiler and GCC deal with loop optimizations and function inlining. These optimizations will improve performance if the code accounts for a significant part of the total program execution. In particular, function inlining has multiple potential benefits. Obviously, it can reduce branch penalties by removing branches on both function call and exit, and potentially also stack usage. Equally importantly, it enables the compiler to optimize over a larger block of code that can lead to better optimizations for value range propagation and elimination of unused code.

However, modifications intended for speed optimizations that increase code size can actually reduce performance because of cache issues. Larger code is less likely to fit in the L1 cache (or indeed the L2 cache) and the performance lost by the additional cache linefills can well outweigh any benefits of the optimization. It is often better to use the armcc `-Ospace` or gcc `-Os` option to optimize for code density rather than speed. Clearly, using Thumb code will also improve code density and cache efficiency.

There are some interesting decisions to be made around function inlining and in some cases human judgment can improve on that of the compiler. A function that is only ever called from one place will always give a benefit if inlined. One might think that inlining very small functions always gives a benefit, but this is not the case. An instance of a tiny function that is called from many places

is likely to be re-used many times within the instruction cache. If the same function is repeatedly inlined, it is much more likely that it will cause a cache miss and also evict other potentially useful code from the cache. The branch prediction logic within Cortex-R series processors is efficient and an unconditional function call and return consumes few cycles, much less than would be used for a cache linefill. You might want to use the GCC function attributes `noinline` or `always_inline` to control such cases.

This is a general problem and not specific to inlining functions. Whenever conditional execution is used and it is lopsided, that is, the expression far more often leads to one result than the other, there is the potential for false static branch prediction and bubbles (a delay in execution of an instruction) in the pipeline. It is usually better to order conditional blocks so that the often-executed code is linear, while the less commonly executed code has to be branched to and does not get pre-fetched unless it is actually used. The GCC attribute `__builtin_expect` used with the `-freorder-blocks` optimization option can help with this.

The performance monitor block of the processor can be used to measure branch prediction rates in code. There are two effects at play here. Correct branch prediction saves clock cycles by avoiding pipeline flushes, but taking fewer conditional branches that skip forward over code can help performance by making more of the program fit within the L1 cache.

15.3.7 Prefetching a memory block access

ARM Cortex-R processors contain sophisticated cache systems and support for instruction prefetching that can hide latencies associated with external memory accesses. The Cortex-R7 processor also includes support for out of order execution that can reduce the memory access latency even further.

However, accesses to the external memory system are usually sufficiently slow that there will still be some penalty. If the processor prefetches instructions or data into the cache before requiring them, then it can hide this latency.


Note

The Cortex-R4 and Cortex-R5 processors have minimal support for out of order execution. Only the Divide instructions and some Floating point operations can complete out of order.

ARM processors provide support for preloading of data, using the `PLD` instruction. The `PLD` instruction is a hint that enables you to request that data is loaded to the data cache in advance of it actually being read or written by the application. The `PLD` operation might generate a cache linefill or a data cache miss, independent of load and store instruction execution, while the core continues to execute other instructions. If supported and used correctly, `PLD` can significantly improve performance by hiding memory access latencies.

In addition to this programmer-initiated prefetch, the core might also support automatic data prefetching. Essentially, the core can detect a series of sequential accesses to memory. When it does, it automatically requests the following cache lines speculatively, in advance of the program actually using them.

In many systems, significant numbers of cycles are consumed initializing or moving blocks of memory, using the `memset()` or `memcpy()` functions. Optimized ARM libraries will typically implement such functions by using Store Multiple instructions, with each store aligned to a cache line boundary.

15.3.8 Branch predictability

Flushing the pipeline has a performance penalty. To avoid unnecessary pipeline flushes the Cortex-R series processors include branch prediction logic.

In compiled C, branching are used for:

- Function calls.
- Function returns.
- Loops.
- Conditional statements such as `if else`.

Some forms of branching are more predictable than others. The `for` loop in the first example below is more predictable than the while loop in the second example. This is because in the second example the number of iterations is dependent on an external condition.

```
Example 1:

  for (i=10; i > 0; i--)
  {
    doSomething();
  }
```

```
Example 2:

  while (TRUE == pMyPeripheral->uiFlagReg)
  {
    doSomething();
  }
```

The purpose of these examples is not to say that you should never write code similar to the second example. Rather, that you should be aware of its effect on branch prediction.

15.4 Source code modifications

Profiling tools enable you to identify code segments or functions that can benefit from optimization and how different compiler options can enable compiler optimizations to our code. We will now consider a variety of source code modifications that can yield faster or smaller code on the ARM.

15.4.1 Loop termination

For loops that have been identified by the profiler, it might be appropriate to have integer loop counters that end at 0 (zero), rather than start from 0 (zero). This is because a compare with zero comes for free with the ADD or SUB instruction used to update the loop counter, whereas a compare with a non-zero value will typically require an explicit CMP instruction.

Replace a loop that counts up to a terminating value:

```
for (i = 1; i<= total; i++)
```

with one that counts down to zero:

```
for (i = total; i != 0; i--)
```

This will remove a CMP instruction from each iteration of the loop.

It is also good practice to use `int` (32-bit) variables for loop counters. This is because the ARM is natively a 32-bit machine. Its ADD assembly language instruction operates on two 32-bit registers. If it carries out an ADD (or other data processing operation) with a smaller quantity, the compiler might insert additional instructions to handle overflow (see also Variable selection).

15.4.2 Loop fusion

This is one of a variety of other possible loop techniques that can be employed either by you, or by an optimizing compiler. It essentially means merging loops that have the same iteration count and no interdependencies.

```
for (i = 0; i < 10; i++)
{
  x[i] = 1;
}
for (j = 0; j < 10; j++)
{
  y[j] = j;
}
```

It is immediately apparent that this can be optimized to:

```
for (i = 0; i < 10; i++)
{
  x[i] = 1;
  y[i] = i;
}
```

It is worth mentioning that this approach can sometimes lead to a reduction in performance because of cache effects such as thrashing, depending on the cache associativity and the addresses of the data being accessed.

15.4.3 Reducing stack and heap usage

In general, it is a good idea to try to minimize memory usage by code. The ARM processor has a register set that provides a relatively limited set of resources for the compiler to keep variables in. When all registers are allocated with currently live variables, additional variables are spilled to the stack, causing memory operations and extra cycles for the code to execute. There are a number of ways available to you, to try to help. A key rule is to try to limit the number of live variables at any one time.

Up to four parameters can be passed in registers to a function. Additional parameters are passed on the stack. It is therefore significantly more efficient to pass four or fewer parameters than to pass five or more. Of course, the ARM registers in question are 32-bits in size and therefore if you pass a 64-bit variable, it will take two of our four register slots. For similar reasons, recursive functions do not typically yield efficient processor register usage. Remember also that non-static C ++ functions also consume one argument slot with the `this` pointer.

15.4.4 Variable selection

ARM integer registers are 32-bit sized and optimal code is therefore produced most readily when using 32-bit sized variables, as this avoids the requirement to provide extra code to deal with the case where a 32-bit result overflows an 8-bit or 16-bit sized variable.

Consider the following code:

```
unsigned int i, j, k;
i = j+k;
```

The compiler would typically emit assembly code similar to:

```
ADD R0, R1, R2
```

If these variables were instead `short` (16-bit) or `char` (8-bit), the compiler must ensure the result does not overflow the halfword or byte.

The same code might be as shown below, for signed halfwords (shorts).

```
ADD      R0, R1, R2
SXTH     R0, R0
```

Or for unsigned halfwords:

```
ADD       R0, R1, R2
BIC       R0, R0, #0x10000
```

This has the effect of clipping the result to the defined size.

Although the compiler can sometimes cope with such things as an incorrect type specification for a loop counter variable, it is generally best to use the correct type in the first place.

15.4.5 Pointer aliasing

If a function has two pointers `pa` and `pb`, with the same value, we say the pointers alias each other. This introduces constraints on the order of instruction execution. If two write accesses that alias occur in program order, they must happen in the same order on the processor and cannot be re-ordered. This is also the case for a write followed by a read, or a read followed by a write. Two read accesses to aliases are safe to re-order. Because any pointer could alias any other pointer in C, the compiler must assume that memory regions accessed through these pointers can overlap, which prevents many possible optimizations. C++ enables more optimizations, as pointer arguments will not be treated as possible aliases if they point to different types.

C99 introduces the `restrict` keyword that specifies that a particular pointer argument does not alias any other. If you know that pointers do not overlap, using this keyword to give the compiler this information can yield significant improvements. However, misusing it can lead to incorrect program function. The restrict keyword qualifies the pointer and not the object being pointed to. This consideration is not specific to the ARM architecture. When using GCC, you can enable the C99 standard by adding `-std=c99` to your compilation flags.

In code that cannot be compiled with C99, use either `__restrict` or `__restrict__` to enable the keyword as a GCC extension.

Consider the following simple code sequence:

```
void foo(unsigned int *ptr1, unsigned int *ptr2, unsigned int *i)
{
  *ptr1 += *i;
  *ptr2 += *i;
}
```

The pointers could possibly refer to the same memory location and this causes the compiler to generate code that is less efficient. In this example, it must read the value `*i` from memory twice, once for each add, as it cannot be certain that changing the value of `*ptr1` does not also change the value of `*i`.

If the function is instead declared as:

```
 void foo(unsigned int *restrict ptr1, unsigned int *restrict ptr2, unsigned int
*restrict i)
```

This means that the compiler can assume that the three pointers might not refer to the same location and optimize accordingly. You must ensure that the pointers never overlap.

15.4.6 Division

As division is slower than multiplication, in performance-critical code it is almost always worth avoiding divides or replacing them with multiplies. This must be done as a trade-off against code maintainability.

Division with a fixed divisor, that is, one that is known at compile time, is faster than dividing two variable quantities.

15.4.7 Extern data

Accessing external variables requires the processor to execute a series of load instructions to acquire the address of the variable through a base pointer and then read the actual variable value. If multiple variables are defined as members of a structure, they can share a base pointer, saving cycles and instructions. It is therefore good practice to define the variables inside the same `struct`.

15.4.8 Inline or embedded assembler

In some cases, it can be a worthwhile optimization to use assembly code, in addition to C. The general principle here is for you to code in a high level language, use a profiler to determine which sections will produce the most benefit if optimized and then inspect the compiler-produced assembly code to look for possible improvements.

If a code section is identified as being a performance bottleneck, don't reach immediately for the assembly language manual. Improvements to the algorithm should first be sought and then compiler optimizations tried before considering use of assembly code. Even then, it is often the case that poor performance is because of cache misses and memory access delays rather than the actual assembly code.

The ARM Compiler, GCC, and most other C compilers use the `-s` flag to tell the compiler to produce assembly code output. The `-fverbose-asm` command line option can also be useful in gcc. Interleaved source and assembler can be produced by the ARM Compiler with the `--interleave` option.

15.4.9 Complex addressing modes

It is often better to avoid complex addressing modes. In cases where the address to be used for a load or store requires a complex calculation, dual-issue of instructions is not possible. Only the addressing mode that uses a base register plus an offset, specified either by a register or an immediate value, with an optional shift left by an immediate value of two is fast. Other, less commonly used, addressing modes can be executed more quickly by splitting into two instructions that might be dual-issued. For example:

```
MOV R2, R1 LSL#3; LDR R2,[R0, R2]
```

can be faster than

```
LDR R2, [R0, R1 LSL #3]
```

LDRH and LDRB have no extra penalty, but LDRSH and LDRSB have a single cycle load-use penalty, but no early forwarding path and can incur additional latency if a subsequent instruction uses the loaded value.

15.4.10 Unaligned access

Unaligned LDRs have an extra cycle penalty compared with aligned loads, but unaligned LDRs that cross cache-lines have many cycles of additional penalty. In general, stores are less likely to stall the system compared to loads. STRB and STRH have similar performance to STR, because of the merging write buffer. Because there are four slots in the load/store unit, more than four consecutive pending loads will always cause a pipeline stall.

15.4.11 Linker optimizations

Some code optimizations can be performed at the link, rather than the compile stage of the build, for example, unused section elimination and linker feedback. Multi-file optimization can be carried out across multiple C files, and unused sections can be removed. Similarly, multi-file compilation enables the compiler to perform optimization across multiple files instead of on individual files.

15.4.12 Floating point operations

The Cortex-R series processors have an option to include an FPU that implements the VFPv3-D16 architecture. This provides support for both single and double precision. However, the implementation is optimized for single precision.

Therefore, to improve performance, use the single precision type in preference to the double precision type where possible.

16. Boot Code

This chapter looks at what the boot code running in an ARM processor based system does. This examines code that runs immediately after the processor comes out of reset, on a bare-metal system. A bare-metal system is one in which code runs without the use of an operating system.

16.1 Booting a bare-metal system

When the processor is reset, it commences execution at the location of the reset vector in the exception vector table. The exception vector table might be either at address `0x00000000` or at address `0xFFFF0000`. The reset handler code must do some or all of the following:

- In a multi-core system, put non-primary cores to sleep.
- Initialize exception vectors.
- Initialize the memory system, including the MPU.
- Initialize core mode stacks and registers.
- Initialize any critical I/O devices.
- Perform any necessary initialization of VFP.
- Enable interrupts.
- Change core mode or state.
- Call the `main()` application.

The first consideration is placement of the exception vector table. You must make sure that it contains a valid set of instructions that branch to the appropriate handlers.

The `_start` directive in the GNU Assembler tells the linker to locate code at a particular address and can be used to place code in the vector table. It is likely that you will want to access the vector table from a TCM during the main program execution, to provide fast access to exception handlers. The system might be configured to copy the boot code from non-volatile memory into the TCM using the slave port interface prior to the processor boot.

When the core comes out of reset the prefetch unit can be stalled while the boot code is copied into a TCM located at the vector table, when the boot code is copied into the TCM the prefetching logic is released and will fetch code directly from the TCM.

Alternatively the vector table and exception handlers might be copied from the external ROM into the TCM as part of the power-on reset initialization code. When the code has been copied into the TCM the TCM is relocated to the vector table base address.

The following shows an example of code that can be placed in the exception vector table.

```
start
  B    Reset_Handler
```

```
    B    Undefined_Handler
    B    SVC_Handler
    B    Prefetch_Handler
    B    Data_Handler
    NOP  @ Reserved vector
    B    IRQ_Handler

  @ FIQ_Handler will follow directly after this table
```

You might then have to initialize stack pointers for the various modes that your application can make use of. The following example shows code which initializes the stack pointers for FIQ and IRQ modes.

```
  LDR     R0, stack_base
@ Enter each mode in turn and set up the stack pointer
  MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
  MOV     SP, R0
  SUB     R0, R0, #FIQ_Stack_Size
  MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
  MOV     SP, R0
```

The next step is to set up the caches, MPU and branch predictors. An example of such code is shown below. You begin by disabling the MPU and caches and invalidating the caches. The example code is for the Cortex-R7 processor.

For the Cortex-R4 and Cortex-R5 processors, data cache invalidation can be done with a single `CP15` instruction, `MCR p15, 0, r0, c15, c5, 0`, but for the Cortex-R7 processor, boot code must explicitly cycle through the lines of the cache and invalidate them.

In the Cortex-R4 and Cortex-R5 processors branch prediction is enabled when the processor comes out of reset. In the Cortex-R7 branch prediction can be safely enabled when the Branch target address cache has been invalidated. Enabling branch prediction will typically improve the performance of the initialization code.

```
  @ Disable MPU and caches
  DSB

  MRC   p15, 0, r1, c1, c0, 0         @ Read Control Register configuration data

  BIC   r1, r1, #0x1                   @ Disable MPU
  BIC   r1, r1, #(0x1 << 12)           @ Disable I Cache
  BIC   r1, r1, #(0x1 << 2)            @ Disable D Cache

  MCR   p15, 0, r1, c1, c0, 0         @ Write Control Register configuration data
  @ Invalidate BTAC
  MCR   p15, 0, r0, c7, c5, 6

  @ Program Flow Prediction Enable
  MOV   r1, #0
  MRC   p15, 0, r1, c1, c0, 0    @ Read Control Register configuration data
  ORR   r1, r1, #(0x1 << 11)      @ Branch Prediction Enable bit
  MCR   p15, 0, r1, c1, c0, 0    @ Write Control Register configuration data


  @ Invalidate L1 Caches

  @ Invalidate Instruction cache
  MOV   r1, #0
  MCR   p15, 0, r1, c7, c5, 0
```

```
  @ Instruction cache Enable prior to Data Cache Invalidation
  DSB
  MRC   p15, 0, r1, c1, c0, 0    @ Read Control Register configuration data
  ORR   r1, r1, #(0x1 << 12)     @ Enable I Cache
  MCR   p15, 0, r1, c1, c0, 0    @ Write Control Register configuration data


  @ Invalidate Data cache
  @ to make the code general purpose, we calculate the
  @ cache size first and loop through each set + way

  MRC   p15, 1, r0, c0, c0, 0          @ Read Cache Size ID
  MOV   r3, #0x1ff
  AND   r0, r3, r0, LSR #13            @ r0 = no. of sets - 1

  MOV   r1, #0                         @ r1 = way counter way_loop

way_loop:
  MOV   r3, #0                         @ r3 = set counter set_loop

set_loop:
  MOV   r2, r1, LSL #30                @
  ORR   r2, r3, LSL #5                 @ r2 = set/way cache operation format
format:
  MCR   p15, 0, r2, c7, c6, 2          @ Invalidate line described by r2
  ADD   r3, r3, #1                     @ Increment set counter
  CMP   r0, r3                         @ Last set reached yet?
  BGT   set_loop                       @ if not, iterate set_loop
  ADD   r1, r1, #1                     @ else, next
  CMP   r1, #4                         @ Last way reached yet?
  BNE   way_loop                       @ if not, iterate way_loop
```

After this, you can program some regions of the MPU, as shown in the example code below:

```
@ region 0: all memory with r/w access for everyone
 MOV     r0,#0
 MCR     p15,0,r0,c6,c2,0  @ region number
 MOV     r0,#0x0           @ base address
 MCR     p15,0,r0,c6,c1,0  @ base addr
 MOV     r0,#0x3f          @ 4GB, all memory
 MCR     p15,0,r0,c6,c1,2  @ size & enable
 MOV     r0,#0x30f         @ write-back cacheable, shareable full access
 MCR     p15,0,r0,c6,c1,4  @ access control
@ region 1: Device Memory for Peripherals
 MOV     r0,#1
 MCR     p15,0,r0,c6,c2,0  @ region number
 MOV     r0,#0xC0000000
 MCR     p15,0,r0,c6,c1,0  @ base addr
 MOV     r0,#0x29          @ 2MB
 MCR     p15,0,r0,c6,c1,2  @ size & enable
 MOV     r0,#0x1301        @ Shareable Device Memory, XN, full access
 MCR     p15,0,r0,c6,c1,4  @ access control

@ Enable MPU and Data Cache
 DSB
 MRC   p15, 0, r1, c1, c0, 0 @ Read Control Register configuration data
 ORR   r1, r1, #(0x1 << 0) @ Enable MPU
 ORR   r1, r1, #(0x1 << 2) @ Enable D Cache
 MCR   p15, 0, r1, c1, c0, 0 @ Write Control Register configuration data
 ISB
```

The TCMs might have to be relocated in the memory map and enabled as part of the boot code. As the following example shows.

```
@ Relocate and enable TCM

 DSB
 ORR r0, #0x00000000        @ set base address to 0x00000000
 MOV r0, #0x1                           @ enable TCM
 MCR p15, 0, r0, c9, c1, 1  @ write TCM region register
 ISB
```

A level 2 cache, if present, and if running without an operating system, might also have to be invalidated and enabled at this point. VFP access must also be enabled.

The next steps will depend on the exact nature of the system. It might be necessary, for example, to:

- Zero-initialize memory that will hold uninitialized C variables.
- Copy the initial values of other variables from a ROM image to RAM
- Set up application stack and heap spaces
- Initialize C library functions
- Call top-level constructors (for C++ code)
- Do other standard embedded C initialization.

For multi-core processors, such as the Cortex-R7 processor, a common approach is to permit a single core within the cluster to perform system initialization. If the same code runs on a different core, it will cause it to enter WFI state and sleep, as described in Power Management. The other core might wake up after CPU0 has initialized the SCU Tag RAMs. The following shows example code that determines which processor it is running on and then either branches to initialization code, if running on CPU0, or goes to sleep otherwise. The SMP OS usually wakes up the secondary cores later.

```
  @ Only CPU 0 performs initialization. Other CPUs go into WFI
  @ to do this, first work out which CPU this is
  @ this code typically is run before any other initialization step

  MRC   p15, 0, r1, c0, c0, 5          @ Read Multiprocessor Affinity Register
  AND   r1, r1, #0x3                    @ Extract CPU ID bits
  CMP   r1, #0
  BEQ   initialize                     @ if we're on CPU0 goto the start

wait_loop:
  @ Other CPUs are left powered-down
  .....
  .....
  .....

initialize:
  @ next section of boot code goes here
```

17. Power Management

Many ARM processors are in battery-powered mobile devices. In such systems, optimization of power usage is a key design constraint. Programmers often spend significant amounts of time trying to save battery life in such systems. Power-saving can also be of concern even in systems that do not use batteries. For example, you might want to minimize energy usage for reduction of electricity costs to the consumer or for environmental reasons.

Built into ARM processors are many hardware design methods aimed at reducing power usage.

Energy usage can be divided into two components - dynamic and static. Both are important. Static power consumption occurs whenever the processor logic or RAM blocks have power applied to them. In general terms, the leakage currents (any current that flows when the ideal current is zero) are proportional to the total silicon area - the bigger the chip, the more the leakage. The proportion of power consumption due to leakage gets significantly higher as we move to more advanced manufacturing process - they are much worse on fabrication geometries of 130nm and below. Dynamic power consumption occurs because of transistors switching and is a function of the processor clock speed and the numbers of transistors that change state per cycle. Clearly, higher clock speeds and more complex processors will consume more power.

Power management aware operating systems dynamically change the power states of cores, balancing the available compute capacity to the current workload, while striving to use the minimum amount of power. Some of these techniques dynamically switch cores on and off, or place them into quiescent states, where they no longer perform computation. This means they consume very little power. The main example is Idle management.

17.1 Idle management

When a core is idle the Operating System Power Management (OSPM) transitions it into a low power state. Typically, a choice of states is available, with different entry and exit latencies, and different levels of power consumption, associated with each state. The state that is used typically depends on how quickly the core is required again. The power states that can be used at any one time might also depend on the activity of other components in an SoC, beside the cores. Each state is defined by the set of components that are clock-gated or power-gated when the state is entered. States are sometimes described as being shallow or deep.

The time required to move from a low power state to a running state, known as the wakeup latency, is longer in deeper states. Although idle power management is driven by thread behavior on a core, the OSPM can place the platform into states that affect many other components beyond the core itself. If the last core in a cluster becomes idle, the OSPM can target power states that affect the whole cluster. Equally, if the last core in a SoC goes idle the OSPM can target power states that affect the whole SoC. The choice is also driven by the usage of other components in the system. A typical example is placing memory in self-refresh when all cores, and any other bus masters, are idle.

The OSPM has to provide the necessary power management software infrastructure to determine the correct choice of state. In idle management, once a core or cluster has been placed into a low power state, it can be reactivated at any time by a processor wakeup event. That is, an event that can wake up a core from a low power state, such as interrupt. No explicit command is required by the OSPM to bring the core or cluster back into operation. The OSPM considers the affected core or cores to be available at all times even if they are currently in a low power state.

17.1.1 Power and clocking

One way you can reduce energy usage is to remove power, that removes both dynamic and static currents (sometimes called power gating) or to stop the clock of the core that removes dynamic power consumption only and can be referred to as clock gating.

ARM processors typically support a number of levels of power management, as follows:

- Standby.
- Retention.
- Power down.
- Dormant mode.

For certain operations, there is a requirement to save and restore state before and after removing power and both the time taken to do this and power consumed by this extra work can be an important factor in software selection of the appropriate power management activity.

The SoC device that includes the core can have additional low power states, with names such as "STOP" and "Deep sleep." These refer to the ability for the hardware Phase Locked Loop (PLL) and voltage regulators to be controlled by power management software.

17.1.2 Standby

In the standby mode of operation, the core is left powered-up, but most of its clocks are stopped, or clock-gated. This means that almost all parts of the processor are in a static state and the only power drawn is because of leakage currents and the clocking of the small amount of logic that looks out for the wake-up condition.

This mode is entered using either the `WFI` (Wait For Interrupt) or `WFE` (Wait For Event) instructions. ARM recommends the use of a Data Sychronization Barrier (`DSB`) instruction before `WFI` or `WFE`, to ensure that pending memory transactions complete before changing state.

If a debug channel is active, it will remain active. The core stops execution until a wakeup event is detected. The wakeup condition is dependent on the entry instruction. For `WFI` an interrupt or external debug request will wake the core. For `WFE`, a number of specified events exist, including another core in the cluster executing the `SEV` instruction. A request from the Snoop Control Unit (SCU) can also wake up the clock for a cache coherency operation in an multi-core system. This means that the cache of a core that is in standby state will continue to be coherent with caches of other cores. A core reset will always force the core to exit from the standby condition.

Various forms of dynamic clock gating can also be implemented in hardware. For example the SCU, GIC, timers, CP15, and instruction pipeline can be automatically clock gated when an idle condition is detected, to save power.

Standby mode can be entered and exited quickly (typically in two-clock-cycles). It therefore has an almost negligible affect on the latency and responsiveness of the core.

To an operating system managing power, a standby state is mostly indistinguishable from a retention state. The difference is evident to an external debugger, and in hardware implementation, but not evident to the idle management subsystem of an operating system.

17.1.3 Retention

The core state, including the debug settings, is preserved in low-power structures, enabling the core to be at least partially turned off. Changing from low- power retention to running operation does not require a reset of the core. The saved core state is restored on changing from low-power retention state to running operation. From an operating system point of view there is no difference between a retention state and standby state, other than method of entry, latency and usage-related constraints. However, from an external debugger point of view the states differ as External Debug Request debug events stay pending and debug registers in the core power domain cannot be accessed.

17.1.4 Power down

In this state the core is powered off. Software on the device has to save all core state, so that it can be preserved over the power-down. Changing from power-down to running operation must include:

- A reset of the core, after the power level has been restored.
- Restoring the saved core state.

The defining characteristic of power down states is that they are destructive of context. This affects all the components that are switched off in a given state, including the core, and in deeper states other components of the system such as the GIC or platform-specific IP. Depending on how debug and trace power domains are organized, in some power-down states one or both of debug and trace context might be lost. Mechanisms must be provided to enable the operating system to perform the relevant context saving and restoring for each given state. Resumption of execution starts at the reset vector, after which each OS must restore its context.

17.1.5 Dormant mode

In dormant mode, the core logic is powered down, but the cache and TCM RAMs are left powered up. Often the RAMs are held in a low-power retention state where they hold their contents but are not otherwise functional. This provides a far faster restart than complete shutdown, as live data

and code persists in the caches. Again, in a multi-core system, individual cores can be placed in dormant mode.

In a multi-core system where individual cores within the cluster are able to go into dormant mode, there is no scope for maintaining coherency while the core has its power removed. Such cores must therefore first isolate themselves from the coherence domain. They will clean all dirty data before doing this and will typically be woken up using another core signaling the external logic to re-apply power.

The woken core must then restore the original core state before rejoining the coherency domain. As the memory state might have changed while the core was in dormant mode, it might have to invalidate the caches anyway. Dormant mode is therefore much more likely to be useful in a single core environment rather than in a cluster. This is because of the additional expense of leaving and rejoining the coherency domain. In a cluster, dormant mode is typically likely to be used only by the last core when the other cores have already been shutdown.

17.2 Assembly language power instructions

ARM assembly language includes instructions that can be used to place the core in a low power state. The architecture defines these instructions as hints - the processor is not required to take any specific action when it executes them. In the Cortex-R processor family, however, these instructions are implemented in a way that shuts down the clock to almost all parts of the core. This means that the power consumption of the processor is significantly reduced - only static leakage currents are drawn, and there is no dynamic power consumption.

The `WFI` instruction has the effect of suspending execution until the processor is woken up by one of the following conditions:

- An IRQ interrupt, even if the CPSR I-bit is set.
- An FIQ interrupt, even if the CPSR F-bit is set.
- An asynchronous abort.
- A Debug Entry request, even if JTAG Debug is disabled.

In the event of the core being woken by an interrupt when the relevant CPSR interrupt flag is disabled, the core will implement the next instruction after `WFI`. On older versions of the ARM architecture, the wait for interrupt function (also called standby mode) was accessed using a CP15 operation, rather than a dedicated instruction.

The `WFI` instruction is widely used in systems that are battery powered. For example, mobile telephones can place the processor in standby mode many times a second, while waiting for you to press a button.

`WFE` is similar to `WFI`. It suspends execution until an event occurs. This can be one the events listed above, or an additional possibility - an event signaled by another core in a cluster. Other cores can signal events by executing the `SEV` instruction. `SEV` signals an event to all cores in a cluster.

18. Debug

Debugging is a key part of software development and is often considered to be the most time consuming (and therefore expensive) part of the process. Bugs can be difficult to detect, reproduce and fix and it can be difficult to predict how long it will take to resolve a defect. The cost of resolving problems grows significantly when the product is delivered to a customer. In many cases, when a product has a small time window for sales, if the product is late, it can miss the market opportunity. Therefore, the debug facilities provided by a system are a vital consideration for any developer.

Many embedded systems using ARM cores have limited input/output facilities. This means that traditional desktop debug methods (such as use of `printf()`) might not be appropriate. In such systems in the past, developers might have used expensive hardware tools like logic analyzers or oscilloscopes to observe the behavior of programs. The cores described in this book have caches and are part of a complex system-on-chip containing memory and many other blocks. There might be no core signals that are visible off-chip and therefore no ability to monitor behavior by connecting up a logic analyzer (or similar). For this reason, ARM systems typically include dedicated hardware to provide wide-ranging control and observation facilities for debug.

18.1 ARM debug hardware

Cortex-R series processors provide hardware features that enable debug tools to provide significant levels of control over core activity and to non-invasively collect large amounts of data about program execution. We can sub-divide the hardware features into two broad classes, invasive and non-invasive.

Invasive debug provides facilities that enable us to stop programs and step through them line by line (either at the C source level, or stepping through assembly language instructions). This can be by means of an external device that connects to the processor using the chip JTAG pins, or (less commonly) by means of debug monitor code in system ROM. JTAG stands for Joint Test Action Group and refers to the IEEE-1149.1 specification, that was originally designed to standardize testing of electronic devices on boards, but is now widely re-used for processor debug connection. A JTAG connection typically has five pins, two inputs, plus a clock, a reset and an output.

The debugger gives the ability to control execution of the program, enabling you to run code to a certain point, halt the processor, step through code and resume execution. We can set breakpoints on specific instructions (causing the debugger to take control when the processor reaches that instruction). These work using one of two different methods. Software breakpoints work by replacing the instruction with the opcode of the `BKPT` instruction. Obviously, these can only be used on code that is stored in RAM, but have the advantage that they can be used in large numbers. The debug software must keep track of where it has placed software breakpoints and what opcodes were originally located at those addresses, so that it can put the correct code back when you want to execute the instruction where the software breakpoint was located. Hardware breakpoints use comparators built into the processor and stop execution when execution reaches the specified address. These can be used anywhere in memory, as they do not require changes to code, but the hardware provides limited numbers of hardware breakpoint units, for example

six in the Cortex-R7 processor. Debug tools can support more complex breakpoints (for example stopping on any instruction in a range of addresses, or only when a specific sequence of events occurs or hardware is in a specific state). Data watchpoints give debugger control when a particular data address or address range is read or written. These can also be called data breakpoints.

On hitting a breakpoint, or when single-stepping, you can inspect and change the contents of ARM registers and memory. A special case of changing memory is code download. Debug tools typically enable you to change your code, recompile and then download the new image to the system.

18.1.1 Single stepping

Single step refers to the ability of the debugger to move through a piece of code, one instruction at a time. The difference between "Step-In" and "Step-Over" can be explained with reference to a function call. If you "Step-Over" the function call, the entire function is executed as one step, enabling you to continue after a function that you do not want to step through. "Step-in" would mean that you instead single step through the function itself.

18.1.2 Debug events

A debug event is some part of the process being debugged that causes the system to notify the debugger. Debug events can be synchronous or asynchronous. Breakpoints, the `BKPT` instruction, and Watchpoints are all synchronous debug events. When any of these events occur, the processor can respond in one of a number of ways:

- It can ignore the debug event.
- It can takes a debug exception.
- It will enter one of two debug modes, depending on the setup of the Debug Status and Control Register (DSCR):
  - Monitor debug mode.
  - Halt Debug mode.

Both of these are examples of invasive debug.

Halt debug mode

In Halt Debug mode, a debug event causes the processor to enter Debug state. The processor is halted and isolated from the rest of the system. This means that the debugger displays memory as seen by the processor, and the effects of memory management and cache operations will become visible.

In Debug state, the processor stops executing instructions from the location indicated by the program counter, and is instead controlled through the external debug interface, in particular using the Debug Instruction Transfer Register (DBGITR). This enables an external agent, such as a debugger, to interrogate processor context and control all subsequent instruction execution. Both the processor and system state can be modified. Because the processor is stopped, no interrupts are handled until execution is restarted by the debugger.

Monitor debug-mode

In Monitor debug-mode, a debug event causes a debug exception to occur, either related to the instruction execution that generates a Prefetch Abort exception, or a data access that generates a Data Abort exception. Both of these must be handled by the software debug monitor. Because the processor is still operating, interrupts can still be serviced.

18.1.3 Semihosting debug

Semihosting is a mechanism that enables code running on an ARM target to use the facilities provided on a host computer running a debugger.

Examples of this might include keyboard input, screen output, and disk I/O. For example, you might use this mechanism to permit C library functions, such as `printf()` and `scanf()`, to use the screen and keyboard of the host. Development hardware often does not have a full range of input and output facilities, but semihosting enables the host computer to provide these facilities.

Semihosting is implemented by a set of defined software instructions that generate an exception. The application invokes the appropriate semihosting call and the debug agent then handles the exception. The debug agent provides the required communication with the host.

The semihosting interface is common across all debug agents provided by ARM. Tools from ARM use `svc 0x123456` (ARM state) or `svc 0xAB` (Thumb) to represent semihosting debug functions.

Of course, outside of the development environment, a debugger running on a host is not normally connected to the system. It is therefore necessary for the developer to re-target any C library functions that use semihosting, for example, by using `fputc()`. This would involve replacing the library code that used an svc call with code that could output a character.

18.2 ARM trace hardware

Non-invasive debug, often called trace in ARM documentation, enables observation of the core behavior while it is executing. It is possible to record memory accesses performed (including address and data values) and generate a real-time trace of the program, seeing peripheral accesses, stack and heap accesses and changes to variables. For many real-time systems, it is not possible to use invasive debug methods. Consider, for example, an engine management system. While you can stop the core at a particular point, the engine will keep moving and you will not be able to do useful debug. Even in systems with less onerous real-time requirements, trace can be very useful.

Trace is typically provided by an external hardware block connected to the core. This is known as an Embedded Trace Macrocell (ETM) or Program Trace Macrocell (PTM) and is an optional part of an ARM processor based system. System-on-chip designers can omit this block from their silicon to reduce costs. These blocks observe, but do not affect core behavior and are able to monitor instruction execution and data accesses.

There are two main problems with capturing trace. The first is that with today's very high processor clock speeds, even a few seconds of operation can mean trillions of cycles of execution. Clearly,

to look at this volume of information would be extremely difficult. The second, related problem is that today's processors can potentially perform one or more 64-bit cache accesses per cycle, and to record both the data address and data values can require a large bandwidth.

This presents a problem in that typically, only a few pins might be provided on the chip and these outputs can be switched at significantly lower rates than the core can be clocked. If the processor generates 100 bits of information every cycle at a speed of 1GHz, but the chip can only output four bits of trace at a speed of 200MHz, then there is a problem. To solve this latter problem, the trace macrocell will try to compress information to reduce the bandwidth required. However, the main method to deal with these issues is to control the trace block so that only selected trace information is gathered. For example, you might trace only execution, without recording data values, or you might trace only data accesses to a particular peripheral or during execution of a particular function.

In addition, it is common to store trace information in an on-chip memory buffer (the Embedded Trace Buffer (ETB)). This alleviates the problem of getting information off-chip at speed, but has an additional cost in terms of silicon area (and therefore price of the chip) and also provides a fixed limit on the amount of trace that can be captured.

The ETB stores the compressed trace information in a circular fashion, continuously capturing trace information until stopped. The size of the ETB varies between chip implementations, but a buffer of 8 or 16KB is typically enough to hold a few thousand lines of program trace.

When a program fails, if the trace buffer is enabled, you can see a portion of program history. With this program history, it is easier to walk back through your program to see what happened before the point of failure. This is particularly useful for investigating intermittent and real-time failures, that can be difficult to identify through traditional debug methods that require stopping and starting the processor. The use of hardware tracing can significantly reduce the amount of time required to find these failures, as the trace shows exactly what was executed, what the timing was and what data accesses occurred.

18.2.1 CoreSight

The ARM CoreSight technology expands on the capabilities provided by the ETM. Again its presence and capabilities in a particular system are defined by the system designer. CoreSight provides a number of extremely powerful debug facilities. It enables debug of multi-processor systems (both asymmetric and SMP) that can share debug access and trace pins, with full control of which processors are being traced at which times. The embedded cross trigger mechanism enables tools to control multiple cores in a synchronized fashion, so that, for example when one core hits a breakpoint, all of the other cores will also be stopped.

Commercial debug tools can use trace data to provide features such as real-time views of processor registers, memory and peripherals, enabling you to step forward and backward through the program execution. Profiling tools can use the data to show where the program is spending its time and what performance bottlenecks exist. Code coverage tools can use trace data to provide call graph exploration. Operating system aware debuggers can make use of trace (and in some cases additional code instrumentation) to provide high level system context information. Here, we list some of the available CoreSight components and give a brief description of their purpose:

Debug Access Port (DAP)

The DAP is an optional part of an ARM CoreSight system. Not every device will contain a DAP. It enables an external debugger to directly access the memory space of the system without having to put the processor into debug state. To read or write memory without a DAP might require the debugger to stop the processor and have it execute Load or Store instructions.The primary function of the DAP is to provide connectivity between an external debug tool (connected via JTAG or SWD) and the CoreSight debug components contained in the SoC, that is, the external debug tool connectivity to the debug APB. The DAP can also provide connectivity to the main SoC system bus(es). This is common and allows a debug tool to access physical system memory. The DAP can also provide connectivity to further JTAG scan chains. This is quite rare and is typically only used to connect legacy ARM cores into a CoreSight system (though is not limited to this use case).

Embedded Cross Trigger (ECT)

The ECT block is a CoreSight component that can be included within in a CoreSight system. Its purpose is to link together the debug capabilities of multiple devices in the system. For example, you can have two cores that run independently of each other. When you set a breakpoint on a program running on one core, it would be useful to be able to specify that when that core stops at the breakpoint, the other one should also be stopped (regardless of which instruction it is currently executing). The Cross Trigger Matrix and Interface within the ECT enable debug status and control information to be propagated between cores and trace macrocells.

AHB Trace Macrocell

The AMBA AHB Trace Macrocell enables the debugger to have visibility of what is happening on the system memory bus. This information is not directly obtainable from the processor ETM, as the core is unable to determine whether data comes from a cache or external memory.

CoreSight Serial Wire

CoreSight Serial Wire Debug gives a 2-pin connection using a Debug Access Port (DAP) that is equivalent in function to a 5-pin JTAG interface.

System Trace Macrocell (STM)

This provides a way for multiple processors (and processes) to perform `printf()` style debugging. Software running on any master in the system is able to access STM channels without having to be aware of use by others, using very simple fragments of code. This enables timestamped software instrumentation of both kernel and user space code. The timestamp information gives a delta with respect to previous events and can be extremely useful. The STM also provides for a number of hardware inputs which can generate trace message upon change of state.

Trace Memory Controller (TMC)

As already described, adding additional pins to a packaged IC can significantly increase its cost. In situations where you have multiple cores (or other blocks capable of generating trace information) on a single device, it is likely that economics preclude the possibility of providing multiple trace ports. The CoreSight Trace Memory Controller can be used to combine multiple trace sources into a single bus. Controls are provided to enable prioritize and select between these multiple input sources. The trace information can be exported off-chip using a dedicated trace port, through the JTAG or serial wire interface or by re-using I/O ports of

the SoC. Trace information can be stored in an ETB or in system memory. A CoreSight Trace Funnel is used when there is more than one trace source. The CSTF combines multiple trace sources into a single bus (the ATB). The TMC has three uses:

- It can be configured to operate as a FIFO to smooth out any burstiness in trace packets.
- It can be configured to operate as a trace buffer itself
- It can be configured to route the ATB either to another trace sink - such as an TPIU or to an area of system memory which is then used as the trace buffer.

You must consult documentation specific to the device they are using to determine what trace capabilities are present and which tools are available to make use of them.

18.3 Debug monitor

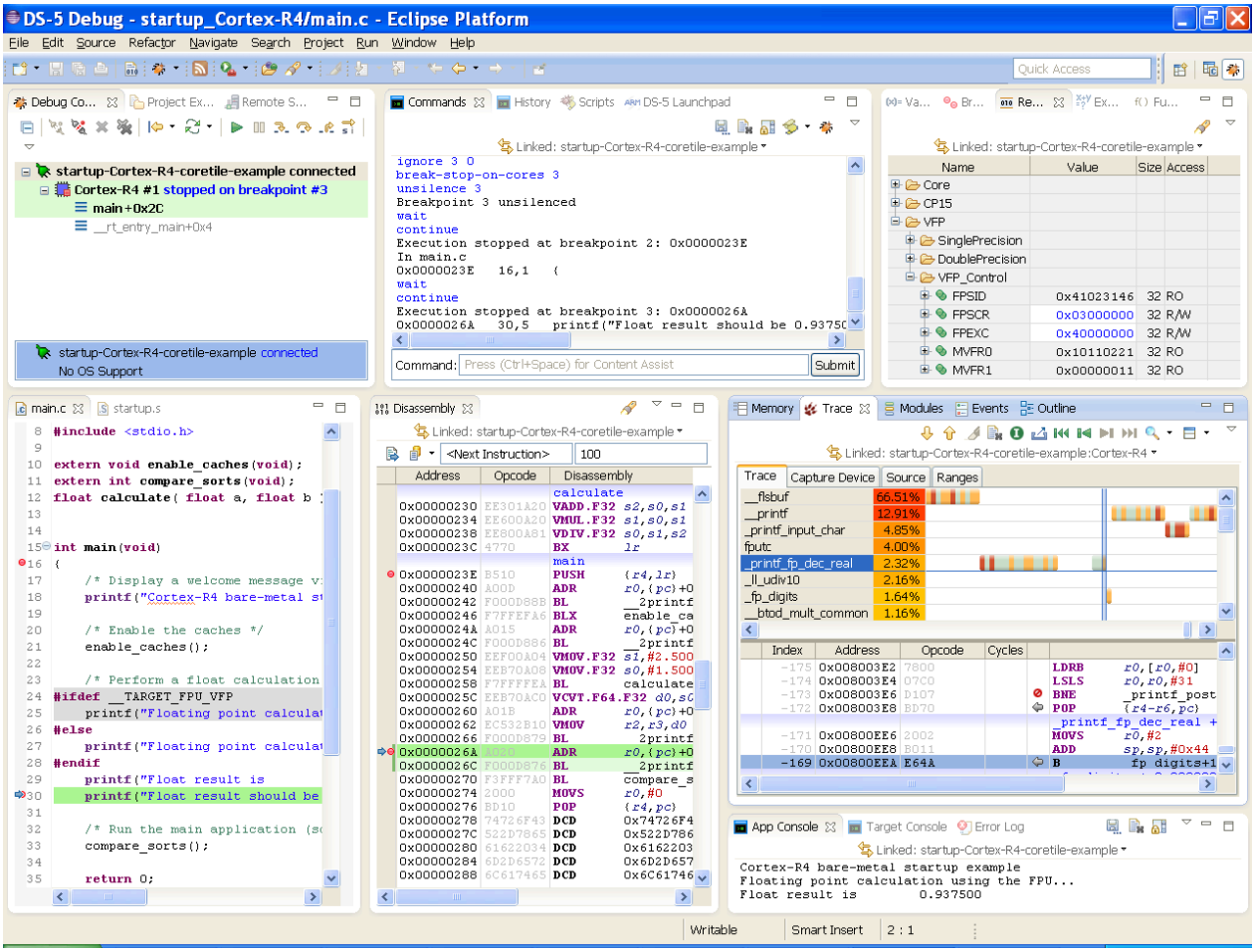
We have seen how the ARM architecture provides a wide range of features accessible to an external debugger. Many of these facilities can also be used by code running on the processor - a so called debug monitor, that is resident on the target system. Monitor systems can be inexpensive, as they might not require any additional hardware. However, they take up memory space in the system and can only be used if the target system itself is actually running. They are of little value on a system that does not at least boot correctly. The breakpoint and watchpoint hardware facilities of the processor are available to a debug monitor. When Monitor mode debug is selected, breakpoint units can be programmed by code running on the ARM processor. If a `BKPT` instruction is executed, or a hardware breakpoint unit matches, the system behaves differently in Monitor mode. Instead of stopping the processor under control of an external hardware debugger, the processor instead takes an abort exception and this can recognize that the abort was generated by a debug event and call the Monitor code.

18.4 ARM DS-5

ARM DS-5 is a professional software development solution suitable for Real-time operating system environments and bare-metal embedded systems based on ARM processor based hardware platforms. DS-5 covers all the stages in development, from boot code and kernel porting to application debug. See http://ds.arm.com/.

ARM DS-5 features an application and kernel space graphical debugger with trace, system-wide performance analyzer, real-time system simulator, and compiler. These features are included in an Eclipse-based IDE.

Figure 18-1: DS-5 Debugger



A full list of the hardware platforms that are supported by DS-5 is available from http://ds.arm.com/supported-devices/.

ARM DS-5 includes the following components:

- Eclipse-based IDE combines software development with the compilation technology of the DS-5 tools. Tools include a powerful C/C++ editor, project manager and integrated productivity utilities such as the Remote System Explorer (RSE), SSH and Telnet terminals.
- DS-5 Compilation Tools include both GCC and the ARM Compiler.
- The DS-5 Debugger, shown in Figure 18-1: DS-5 Debugger on page 214, together with a supported debug target.

  It gives complete control over the flow of program execution to quickly isolate and correct errors. It provides comprehensive and intuitive views, including synchronized source and disassembly, call stack, memory, registers, expressions, variables, threads, breakpoints, and trace.

  A set of example projects are provided, including bare-metal startup code examples for the range of ARM processors, including Cortex-R4, Cortex-R5 and Cortex-R7 processors.

18.4.1 DS-5 debug and trace

DS-5 Debugger takes care of downloading and connecting to the debug hardware, such as DSTREAM debug and trace unit. Developers must specify the platform and the IP address. This reduces a complex task using several applications and a terminal to a couple of steps in the IDE.

In addition, DS-5 Debugger supports ARM CoreSight to provide non-intrusive program trace that enables you to review instructions (and the associated source code) as they have occurred. It also provides the ability to debug time-sensitive issues that would otherwise not be picked up with conventional intrusive stepping techniques. The DS-5 Debugger currently uses DSTREAM to capture trace on the Embedded Trace Buffer (ETB) a small (typically 8K) on-chip trace buffer. This buffer is accessible from DSTREAM and many third party probes. DSTREAM also supports connection to the Trace Port Interface Unit (TPIU) for collection of up to 4GB of trace data.

The DS-5 Debugger provides a powerful tool for debugging applications on both hardware targets and models using ARM architecture-based processors. You can have complete control over the flow of the execution so that you can quickly isolate and correct errors.

The following features are provided in the DS-5 Debugger:

- Loading images and symbols.
- Running images.
- Breakpoints and watchpoints.
- Source and instruction level stepping.
- Controlling variables and register values.
- Viewing the call stack.
- Support for handling exceptions and Linux signals.
- Debug of multi-threaded Linux applications.
- Debug of Linux kernel modules, boot code and kernel porting.

The debugger supports a comprehensive set of DS-5 Debugger commands that can be executed in the Eclipse IDE, script files, or a command-line console. Python scripting is also supported. In addition, there is a small subset of CMM-style commands sufficient for running target initialization scripts.

DS-5 Debugger supports bare-metal debug using JTAG, Linux application debug using `gdbserver`, Linux kernel debug using JTAG, and Linux kernel module debug using JTAG. Debug and trace support is included for bare-metal SMP systems, including cross-triggering and core-dependent views and breakpoints and ETM/PTM trace. This support is described in the following sections.

18.4.2 Debugging a multi-threaded applications using DS-5

DS-5 Debugger tracks the current thread using the debugger variable, `$thread`. You can use this variable in print commands or in expressions. Threads are displayed in the Debug Control view with

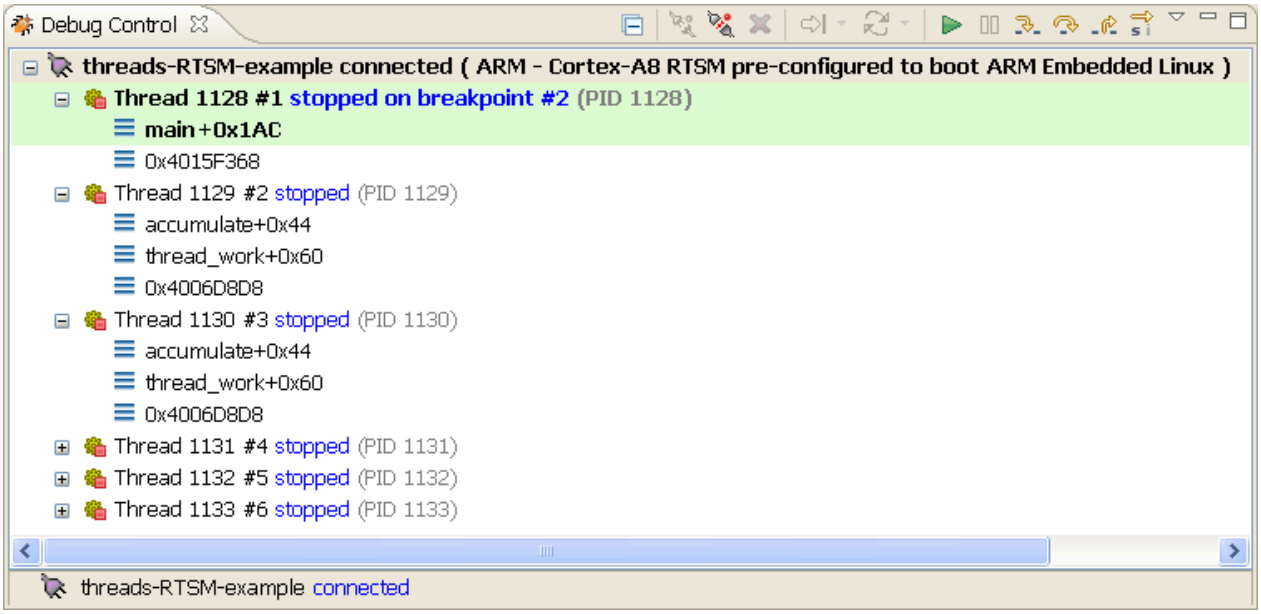
a unique ID that is used by the debugger and a unique ID from the Operating System (OS). For example:

```
Thread 1 (OS ID 1036)
```

where `Thread 1` is the ID used by the debugger and `OS ID 1036` is the ID from the OS.

A separate call stack is maintained for each thread and the selected stack frame is shown in bold text. All the views in the DS-5 Debug perspective are associated with the selected stack frame and are updated when you select another frame.

Figure 18-2: Threading call stacks in the DS-5 Debug Control view



18.4.3 Trace support in DS-5

DS-5 enables you to perform trace on your application or system. You can capture in real-time a historical, non-intrusive trace of instructions. Tracing is a powerful tool that enables you to investigate problems while the system runs at full speed. These problems can be intermittent, and are difficult to identify through traditional debugging methods that require starting and stopping the processor. Tracing is also useful when trying to identify potential bottlenecks or to improve performance-critical areas of your application.

Before the debugger can trace function executions in your application you must ensure that:

- You have a debug hardware agent, for example, an ARM DSTREAM unit with a connection to a trace stream.
- The debugger is connected to the debug hardware agent.

Trace view

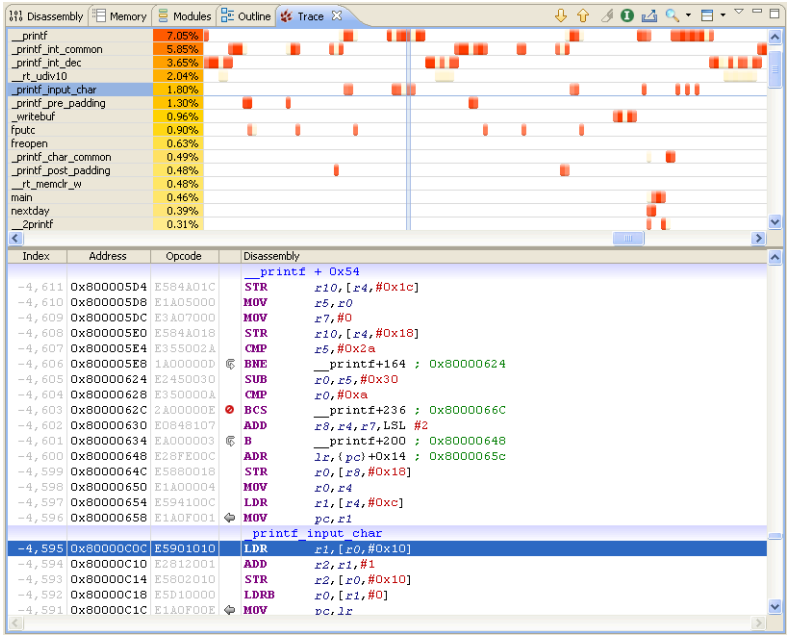
When the trace has been captured the debugger extracts the information from the trace stream and decompresses it to provide a full disassembly, with symbols, of the executed code.

This view shows a graphical navigation chart that displays function executions with a navigational timeline. In addition, the disassembly trace shows function calls with associated addresses and if selected, instructions. Clicking on a specific time in the chart synchronizes the disassembly view.

In the left-hand column of the chart, percentages are shown for each function of the total trace. For example, if a total of 1000 instructions are executed and 300 of these instructions are associated with `myFunction()` then this function is displayed with 30%.

In the navigational timeline, the color coding is a "heat" map showing the executed instructions and the amount of instructions each function executes in each timeline. The darker red color showing more instructions and the lighter yellow color showing less instructions. At a scale of 1:1 however, the color scheme changes to display memory access instructions as a darker red color, branch instructions as a medium orange color, and all the other instructions as a lighter green color.

Figure 18-3: DS-5 Debugger Trace view



Trace-based profiling

Based on trace data received from a trace buffer such as the ETB, The DS-5 Debugger can generate timeline charts with information to help developers to quickly understand how their software executes on the target and which functions are using the processor the most. The timeline offers various zoom levels, and can display a heat-map based on the number of instructions per time unit or, at its highest resolution, provide per-instruction visualization color-coded by the typical latency of each group of instructions.